

CT248: Introduction to Modelling

1. MATLAB Overview

Prof. Jim Duggan,
School of Computer Science
National University of Ireland Galway.
<https://github.com/JimDuggan/CT248>

Course Overview

- Lectures
 - Monday 5-6
 - Wednesday 12-1
- Continuous Assessment (30%)
 - Labs 4-6 Thursdays, Starting Week 3, with assignments
 - Lab Exams (1 mid-course and 1 end of course)



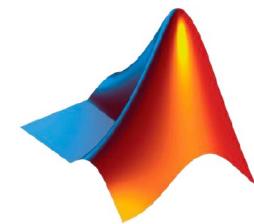
MATLAB

- MATLAB is a powerful technical computing system for handling scientific and engineering calculations.
- MATrix LABoratory
- Designed to make matrix computations particularly easy

<https://uk.mathworks.com/academia/tah-portal/national-university-of-ireland-galway-31528171.html>

What is MATLAB?

- High-level language
- Interactive development environment
- Used for:
 - Numerical computation
 - Data analysis and visualization
 - Algorithm development and programming
 - Application development and deployment



Download MATLAB (Campus Wide License)

The screenshot shows the MathWorks website for National University of Ireland Galway. At the top, there's a navigation bar with links for 'Get Software', 'Learn MATLAB', 'Teach with MATLAB', and 'What's New'. Below the navigation is a banner with the text 'MATLAB Access for Everyone at National University of Ireland Galway' and the NUI Galway logo. To the right of the banner is a section titled 'MATLAB and Simulink are' with a bulleted list: 'used in 100,000+ companies from market leaders to startups' and 'referenced in 4 million+ research citations'. Below this is a question 'Where will MATLAB and Simulink take you?'. On the left side of the main content area is a photograph of several students in a lab setting. On the right side, there's a screenshot of a MATLAB workspace window showing code related to image classification.

<https://uk.mathworks.com/academia/tah-portal/national-university-of-ireland-galway-31528171.html>

Introduction

1. Variables, operators and expressions
2. Basic input and output
3. Repetition (for)
4. Relational Operators and Decisions (if)
5. Arrays (including vectors and Matrices)
6. *Review: Matrix Maths*



(1) Variables

- Variables are fundamental to programming
- A variable name must comply with two rules:
 - It may consist only of letters a-z, digits 0-9, and the underscore (_)
 - It must start with a letter
- MATLAB only remembers the first 63 characters

Case Sensitivity

- MATLAB is case-sensitive
- Command and function names are case sensitive
- “Camel caps” or underscores can be used for longer names
- The semi-colon prevents the variable value from being displayed

```
carSpeed = 20;
```

```
car_speed = 20;
```

<https://sites.google.com/site/matlabstyleguidelines/naming-conventions>

The MATLAB Workspace

- All the variables created during a session remain in the workspace until you *clear* them.
- The command *who* lists all the names in your workspace
- The command *whos* lists the size of each variable

```
clear;

mph = input ('Please enter the
speed in mph : ');

kph = mph * 1.6;

fprintf("%d mph = %d
kph\n", mph, kph);
```

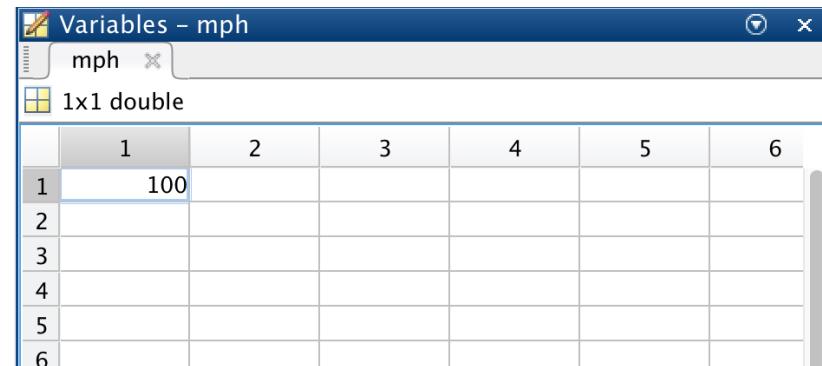
```
>> whos
  Name      Size            Bytes  Class       Attributes
  kph      1x1              8  double
  mph      1x1              8  double
```



Explanation

- Each variable occupies 8 bytes of storage
- 1×1 is a scalar
- The Workspace browser on the desktop provides a useful visual representation of the workspace
- Workspace values can be changed using the array editor

```
>> whos
Name      Size            Bytes  Class       Attributes
kph       1x1              8  double
mph       1x1              8  double
```



Expressions

- An expression is a formula consisting of variables, numbers, operators and function names
- It is evaluated when you enter it at the MATLAB prompt
- Note that MATLAB uses the function ans to return the last expression to be evaluated but not assigned to a variable.

```
>> 2 * pi  
  
ans =  
  
6.2832  
  
>>  
>> ans  
  
ans =  
  
6.2832
```

Statements

- MATLAB statements are frequently of the form
 $variable = expression$
 $s = u*t - g/2 * t.^2$
- This is an example of an assignment statement, with the value of the expression assigned to the variable on the LHS
- A semi-colon at the end suppresses output
- Statements that are too long for a line can be extended with an ellipsis of at least three dots
- Statements on the same line can be separated by commas or semicolons.

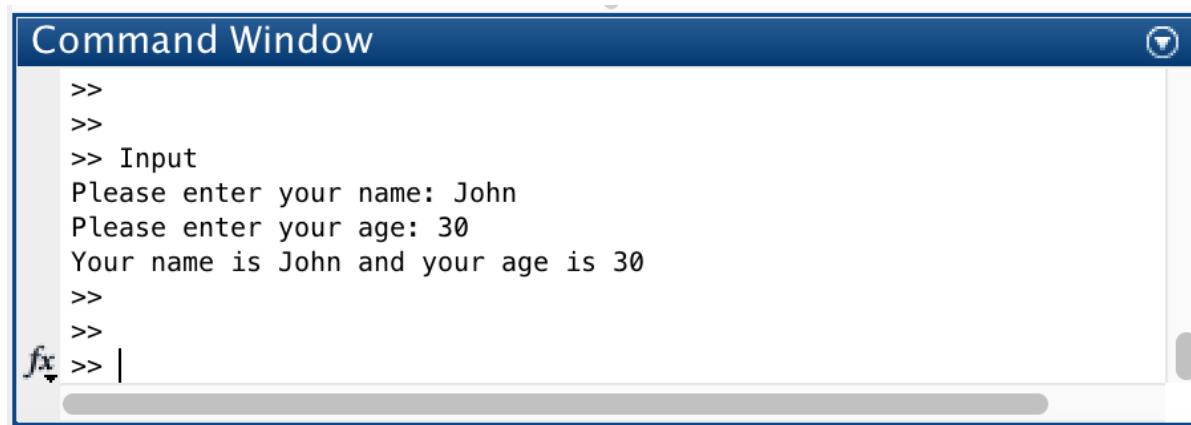
(2) Input and Output

```
clear;

% 's' forces the input to be a character string
name = input ('Please enter your name: ', 's');

age = input ('Please enter your age: ');

fprintf("Your name is %s and your age is %d\n",name,age);
```



A screenshot of the MATLAB Command Window. The window title is "Command Window". The command history shows several inputs starting with ">>". The user has entered the following code:

```
>>
>>
>> Input
Please enter your name: John
Please enter your age: 30
Your name is John and your age is 30
>>
>>
fx >> |
```

The window shows the user's input ("Input") and the resulting output ("Please enter your name: John", "Please enter your age: 30", "Your name is John and your age is 30"). The prompt "fx >> |" indicates where the user can type the next command.

Output: the disp statement

- The general form of disp for a numeric variable is
disp(variable)
- To display more than one variable, embed variables into an array
- All components of a MATLAB array must be the same type.

```
>> disp('Hello world')
Hello world
>> disp('Hello "world"')
Hello 'world'
>>
>> x = 2
x =
2.00
>> x = 2;
>>
>> disp(['The answer is ', num2str(x)])
The answer is 2
>>
>> disp([2 3 x]);
2.00    3.00    2.00
```

Convert F to C

Input – Process - Output

```
% Script file for converting temperatures from F to C  
% Taken from Essential MATLAB (D. Valentine) p 42  
  
% Step 1: Get the input  
F = input ('Enter the temperature in degrees F: ');  
  
% Step 2: Convert to C  
C = (F - 32) * 5 / 9;  
  
% Step 3: Display the result  
fprintf("The temperature of %f (F) is %f (C)\n",F,C);
```



Arithmetic Operators

- The evaluation of an expression is achieved by means of arithmetic operators
- The arithmetic operations on two scalar constants /variables is shown
- Left division seems curious, however matrix left division has an entirely different meaning

Operation	Algebraic Form	MATLAB
Addition	$a + b$	$a + b$
Subtraction	$a - b$	$a - b$
Multiplication	$a \times b$	$a * b$
Right Division	a/b	a / b
Left Division	$b \backslash a$	$b \backslash a$
Power	a^b	$a ^ b$

Precedence	Operator
1	Parentheses (round brackets)
2	Power, left to right
3	Multiplication & Division, left to right
4	Addition and Subtraction, left to right



Challenge 1.1

Evaluate each of these expressions.

```
a1 = 1 + 2 * 3;
```

```
a2 = 4 / 2 * 2;
```

```
a3 = 1 + 2 / 4;
```

```
a4 = 2 * 2 ^ 3;
```

```
a5 = 2 ^ (1+2)/3;
```



(3) Repetition – for statement

```
for i = 1:5  
    disp(i);  
end
```

1.00
2.00
3.00
4.00
5.00

The for loop repeats statements a specific number of times, starting with $i = a$ and ending with $i = b$, incrementing i by 1 each iteration of the loop.

The number of iterations will be $b - a + 1$.



The basic for construct

```
for index = j:k  
    statements;  
end
```

```
for index = j:m:k  
    statements;  
end
```

- $j:k$ is a vector with elements $j, j+1, j+2, \dots, k$
- $j:m:k$ is a vector with elements $j, j+m, j+2m, \dots$, such that the final element does not exceed k
- $index$ must be a variable
- Loop statements should be indented



Challenge 1.2

- Write a program that takes in n numbers and displays the average



```
clear;

sum = 0;

n = input ('How many numbers : ');

for i = 1:n
    num = input ('Please enter a number : ');
    sum = sum + num;
end

avr = sum/n;

fprintf("The average of the numbers is %f\n",avr);
```



(4) Relational Operators

- Relational operators form a logical expression (condition) that is either true or false
- The basis for decision logic

```
>> 10 > 9           >> 10 == 9
```

```
ans =               ans =
```

logical

1

logical

0

Relational Operator	Meaning
<	Less than
<=	Less than or equal
==	Equal
~=	Not equal
>	Greater than
>=	Greater than or equal

The one-line if statement

- if condition; statements; end

```
num = input('Enter a number: ');

if num < 0; disp('Negative number'); end
```



Challenge 1.3

- The following statements all assign logical values to the variable x. See can you correctly determine the value of x in each case, before checking the answer in MATLAB

`x = 3 > 2`

`x = -4 <= -3`

`x = 1 < 1`

`x = 2 ~= 2`

`x = 3 == 3`



The if-else construct

```
if condition  
    statementsA  
Else  
    statementsB  
end
```

```
num = input('Enter a number: ');  
  
if num >= 0  
    disp([num2str(num) ' is positive'])  
else  
    disp([num2str(num) ' is negative'])  
end
```



elseif Construct

```
if condition1
    statementsA
elseif condition2
    statementsB
elseif condition3
    statementsC
:::
else
    statementsE
end

age = input('Please enter your age: ');
if age >= 65
    discount = 0.25;
elseif age < 18
    discount = 0.10;
else
    discount = 0.0;
end

disp(['Age = ' num2str(age) ...
    ' Discount = ' num2str(discount)])
```



(5) Matrix – A Mathematical Definition

- In linear algebra, a matrix is a rectangular grid of numbers arranged into *rows* and *columns*
- An $r \times c$ matrix has r rows and c columns
- Here is an example of a 2×3 matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$



<http://samples.jbpub.com/9781556229114/chapter7.pdf>

Matrices in MATLAB

- In MATLAB, a matrix is a rectangular object consisting of rows and columns
- A vector is a special type of Matrix (having only one row or one column)
- MATLAB handles vectors and matrices in the same way



Initialising Vectors

- Rules:

- Elements in the list must be enclosed in square brackets
- Elements in the list must be separated by either spaces or by commas

```
>> x = [1 2 3 4 5]  
  
x =  
1 2 3 4 5  
  
>>  
>> size(x)  
  
ans =  
1 5
```



Sample operations on vectors

```
>> x  
x =  
1 2 3 4 5  
  
>>  
>> sum(x)  
  
>>  
>> x(1)  
ans =  
1  
  
>> mean(x)  
  
>> x(1:3)  
ans =  
1 2 3
```



Challenge 1.4

- Can you work out what c will be?

```
a = [1 2 3];  
b = [4 5]  
c = [a -b];
```



Initialising Vectors: The colon operator

```
>> x = 1:10
```

```
x =
```

```
1 2 3 4 5 6 7 8 9 10
```

```
>> x = 1:.5:10
```

```
x =
```

Columns 1 through 15

```
1.0000 1.5000 2.0000 2.5000 3.0000 3.5000 4.0000 4.5000 5.0000 5.5000  
6.0000 6.5000 7.0000 7.5000 8.0000
```

Columns 16 through 19

```
8.5000 9.0000 9.5000 10.0000
```



Transposing Vectors

- Vectors generated so far are row vectors
- Column vectors may be needed for matrix operations
- The single quote can be used to transpose a vector

```
>> v
```

```
v =
```

```
1 2
```

```
>>
```

```
>> v'
```

```
ans =
```

```
1
```

```
2
```



Matrices

- May be thought of as a table consisting of rows and columns
- Has comprehensive support with functions and operators

```
>> a = [1 2 3; 4 5 6]
```

```
a =
```

```
1 2 3  
4 5 6
```

```
>>
```

```
>> a'
```

```
ans =
```

```
1 4  
2 5  
3 6
```



Subscripts to access matrix elements

- Individual elements are usually referenced with two subscripts (row#, column#)

```
clear;
```

```
A = [1 2 3; 4 5 6; 7 8 9];
```

```
A(1,1)
```

```
A(1,2)
```

```
A(2,1)
```

```
A(2);
```



Using : with subscripts

- Using the colon (:) operator in place of a subscript denotes all the elements in the corresponding row or column.
- $A(3,:)$ means all the elements in the third row
- $A(:,3)$ means all the elements in the third column

```
clear;  
A = [1 2 3; 4 5 6; 7 8 9];  
% extract the full row 3  
A(3,:);  
  
% extract the column 3  
A(:,3);  
  
% extract a subset of the matrix  
A(1:2, 2:3);
```



Matrix Functions

Function	Description
eye	Identify matrix
linspace	Vector with linearly spaced elements
ones	Matrix of ones
rand	Uniformly distributed random numbers and arrays
randn	Normally distributed random numbers and arrays
zeros	Matrix of zeros
det	Determinant
eig	Eigenvalues and eigenvectors
expm	Matrix exponential
inv	Matrix inverse
trace	Sum of diagonal elements
{}\backslash and /	Linear equation solution



Challenge 1.5



- Simulate the roll of 2 dice 10 times using the randi function (an $n \times 2$ array). Call rng(100) before to ensure replication of results.
- Sum both of the outcomes for each throw, using the colon (:) operator to extract all rows and 1 column from a matrix.
- Find the number of 7s in the outcomes. Use vectorisation to store the comparison in a new vector (e.g. Try sum == 7 and check the outcome)
- Check the proportion of these as compared to the expected outcome (statistics theory).
- Try the experiment for 10, 100, 1000 and 10000 rolls

Challenge 1.6

- Write a program that accepts a stream of numbers (-1 to terminate the stream)
- Calculate the minimum, maximum and average
- Make use of the *while* statement



(6) Matrix Maths

- In linear algebra, a matrix is a rectangular grid of numbers arranged into *rows* and *columns*
- An $r \times c$ matrix has r rows and c columns
- Here is an example of a 2×3 matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$



<http://samples.jbpub.com/9781556229114/chapter7.pdf>

Matrix Name and elements

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

- A is a 3 by 3 matrix
- Uppercase letters usually used for variables that are matrices
- a_{ij} denotes the element in A at row i and column j

Square Matrices

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

- Matrices with the same number of rows and columns are called *square matrices*
- The diagonal elements of a square matrix are those elements where the row and column index are the same (a_{11}, a_{22}, a_{33})

Diagonal Matrix

$$B = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 20 & 0 \\ 0 & 0 & 10 \end{pmatrix}$$

- If all non-diagonal elements in a matrix are zero, then the matrix is a diagonal matrix.

Identity Matrix

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- A special diagonal matrix is the *identity matrix*.
- This is an $n \times n$ matrix with ones on the diagonal and zeros elsewhere
- It's special because its the multiplicative identify element for matrices

Vectors as Matrices

- Matrices may have any positive number of rows and columns, including one
- Matrices with one row or one columns are vectors
- A vector of dimension n can be viewed as either:
 - $1 \times n$ matrix (row vector)
 - $n \times 1$ matrix (column vector)

Transposition

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}^T = \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}$$

- Consider a matrix M with dimensions $r \times c$
- The transpose of M (M^T) is the $c \times r$ matrix where the columns are formed from the rows of M .
- $M^T_{ij} = M_{ji}$
- For vectors, transposition turns row vectors into column vectors and vice-versa

Multiplying a Matrix with a Scalar

$$kA = k \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} ka_{11} & ka_{12} & ka_{13} \\ ka_{21} & ka_{22} & ka_{23} \\ ka_{31} & ka_{32} & ka_{33} \end{pmatrix}$$

- A matrix A may be multiplied with a scalar k, resulting in a matrix of the same dimension as A
- The multiplication takes place in a straightforward fashion, each element in the new matrix is the product of k with the corresponding element

Multiplying Two Matrices

$$\begin{pmatrix} ? & ? \\ ? & ? \\ ? & ? \end{pmatrix} \begin{pmatrix} ? & ? & ? \\ ? & ? & ? \end{pmatrix} = \begin{pmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{pmatrix}$$

3×2 2×3 3×3

- An $r \times n$ matrix (A) can be multiplied by a $n \times c$ matrix (B)
- The result (AB) is a $r \times c$ matrix



Example from MATLAB

A

1	2
3	4
5	6

B

10	20	30
40	50	60

C = A * B

90	120	150
190	260	330
290	400	510

- Let matrix C be the $r \times c$ product AB of the $r \times n$ matrix A with the $n \times c$ matrix B.
- Each element c_{ij} is equal to the vector dot product of row i of A with column j of B

2x2 Example

$$AB = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$AB = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

Other information

- Matrix multiplication is not commutative
 $AB \neq BA$
- Matrix multiplication is associative
 $(AB)C = A(BC)$
- Matrix multiplication also associates with multiplication by a scalar or vector
 $(kA)B = k(AB) = A(kB)$
 $(vA)B = v(AB)$



Multiplying a Vector and a Matrix

$$(x \ y \ z) \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = (xa_{11}ya_{21}za_{31} + xa_{12}ya_{22}az_{32} + xa_{13}ya_{23}ya_{33})$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} xa_{11} + ya_{12} + za_{13} \\ xa_{21} + ya_{22} + az_{23} \\ xa_{31} + ya_{32} + za_{33} \end{pmatrix}$$

- We can multiply a vector and a matrix using the general rules discussed.
- However, the choice of a row or column vector is important

CT248: Introduction to Modelling

2. Logical Vectors and Arrays

Prof. Jim Duggan,
School of Computer Science
National University of Ireland Galway.
<https://github.com/JimDuggan/CT248>

Overview

- Element-wise operators
- Logical Vectors
- Matrices



Matrix Maths – See Lecture 1

- In linear algebra, a matrix is a rectangular grid of numbers arranged into *rows* and *columns*
- An $r \times c$ matrix has r rows and c columns
- Here is an example of a 2×3 matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$



<http://samples.jbpub.com/9781556229114/chapter7.pdf>

Multiplying a Matrix with a Scalar

$$kA = k \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} ka_{11} & ka_{12} & ka_{13} \\ ka_{21} & ka_{22} & ka_{23} \\ ka_{31} & ka_{32} & ka_{33} \end{pmatrix}$$

- A matrix A may be multiplied with a scalar k, resulting in a matrix of the same dimension as A
- The multiplication takes place in a straightforward fashion, each element in the new matrix is the product of k with the corresponding element

Example from MATLAB

A

1	2
3	4
5	6

B

10	20	30
40	50	60

C = A * B

90	120	150
190	260	330
290	400	510

- Let matrix C be the $r \times c$ product AB of the $r \times n$ matrix A with the $n \times c$ matrix B.
- Each element c_{ij} is equal to the vector dot product of row i of A with column j of B

2x2 Example

$$AB = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$AB = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

(1) Element-Wise Operations

- Matlab is expecting that dimensions of our matrices agree in some linear algebra sense.
- But what we want is to apply our operation to each element of the array.
- The “dot operator” is used for this

```
>> A
```

```
A =
```

```
2 4 6 8
```

```
>>
```

```
>> 1/A
```

```
Error using /
```

```
Matrix dimensions must agree.
```

```
>>
```

```
>> 1./A
```

```
ans =
```

```
0.5000 0.2500 0.1667 0.1250
```

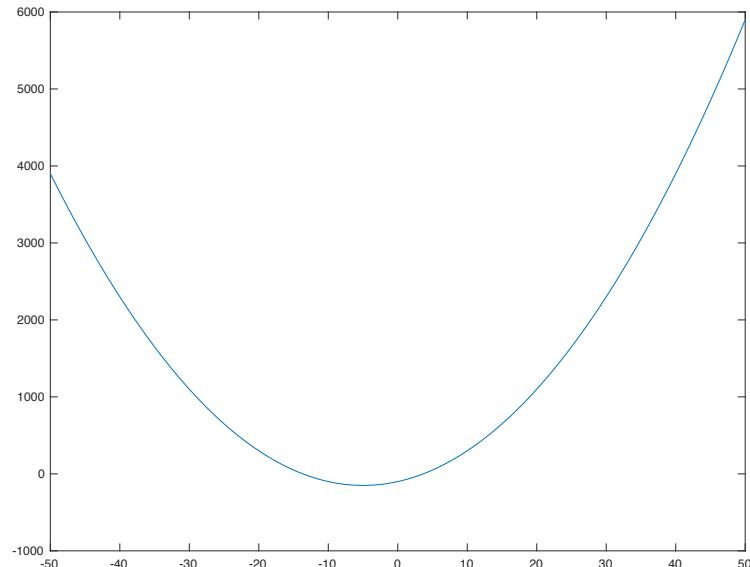
https://math.boisestate.edu/~calhoun/teaching/matlab-tutorials/lab_31/html/lab_31.html

Rules for the dot operator

- **Multiplication:** If both expressions on either side of the multiplication symbol are arrays, then use the `.*` operator. If one of the expressions is a scalar, then no dot is needed.
- **Division:** If the numerator is a scalar and the denominator is an array, use the `./` operator. If both the numerator and the denominator are arrays, also use the `./` operator. If the numerator is an array, and the denominator is a scalar, then no dot is needed.
- **Exponentiation:** If either the base or the power (or both) is an array, use the `.^` operator. If neither is an array, then no dot is needed.
- **Addition and subtraction:** Dots are never used and are not allowed.

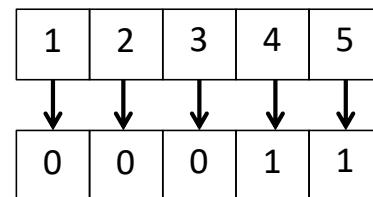
Challenge 2.1

- Generate the following plot for a quadratic equation. $a = 2$, $b = 20$, $c = -100$.
- Use `plot(x,y)`



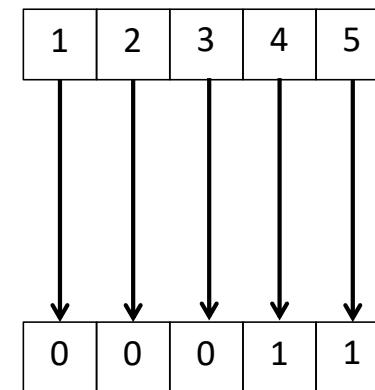
(2) Vectors and Logical Expressions

- When a vector is involved in a logical expression, the comparison is carried out element-by-element
- If the comparison is true, the resulting vector (a logical vector) has a 1 in the corresponding position, otherwise it has a 0



Consider the following code

```
>> v = 1:5  
  
v =  
  
1 2 3 4 5  
  
>>  
>> v > 3  
  
ans =  
  
1×5 logical array  
  
0 0 0 1 1
```



Logical Operators

Operator	Meaning
\sim	NOT
$\&$	AND
$ $	OR

lex1	lex2	\sim lex1	lex1 & lex2	lex1 lex2	$\text{xor}(\text{lex1}, \text{lex2})$
F	F	T	F	F	F
F	T	T	F	T	T
T	F	F	F	T	T
F	F	F	T	T	F



Operator Precedence (update)

Precedence	Operators
1.	$()$
2.	$\wedge . \wedge ' '$ (pure transpose)
3.	$+ \text{ (unary plus)} - \text{ (unary minus)} \sim \text{ (NOT)}$
4.	$* / \backslash . * . / . \backslash$
5.	$+ \text{ (addition)} - \text{ (subtraction)}$
6.	$:$
7.	$> \geq < \leq == \sim=$
8.	$\& \text{ (AND)}$
9.	$ \text{ (OR)}$



Subscripting with logical vectors

- A logical vector v may be a subscript of another vector x
- Only the elements of x corresponding to 1s in v are returned
- x and v must be the same size
- The function `logical(v)` returns a logical vector, with elements which are 1 or 0 according as the elements of v are non-zero or 0.

Example of subscripting...

```
v = 1:5;  
  
lv = v > 3;  
  
disp('Using logical vector to filter v');  
disp(v);  
disp(lv);  
disp(v(lv));
```

```
>> LogicalVector  
Using logical vector to filter v
```

```
1 2 3 4 5
```

```
0 0 0 1 1
```

```
4 5
```


Using the logical() function

```
v = 1:5;  
  
lv = logical([0 1 0 1 0]);  
  
disp('Using logical vector from logical() to filter v');  
disp(v);  
disp(lv);  
disp(v(lv));
```

```
>> LogicalVector2  
Using logical vector from logical() to filter v  
1 2 3 4 5
```

```
0 1 0 1 0
```

```
2 4
```


Logical Functions

- MATLAB has a number of useful logical functions that operate on scalars, vectors and matrices
 - `any(x)` returns the scalar 1 (true) if any element of x is non-zero.
 - `all(x)` returns the scalar 1 if all the elements of x are non-zero
 - `exists('a')` returns 1 if a is a workspace variable
 - `find(x)` returns a vector containing the subscripts of the non-zero (true) elements of x

Sample code

```
v = [1 2 3 4 3 4 5 0];  
  
disp(v);  
disp(any(v));  
disp(all(v));  
disp(find(v==3));
```

```
>> log_functions  
1 2 3 4 3 4 5 0  
  
1  
  
0  
  
3 5
```



Challenge 2.2

- Write code that removes all the duplicated values in a vector



Challenge 2.3

- For a vector of 100 random numbers, filter out all those that are less than the mean
- Build on this to create a function called `partition`, which takes a vector and splits it into two vectors, one containing all numbers less than the mean, the other containing all numbers greater than or equal to the mean.

(3) Matrices

- MATLAB – MATrix LABoratory
- The word matrix has two distinct meanings:
 - An arrangement of data in rows or columns e.g. a table
 - A mathematical object, for which particular mathematical operations are defined (e.g. matrix multiplication)



Creating matrices

- Bigger matrices can be constructed from smaller ones

```
a = [1 2; 3 4];           >> CreateM  
x = [5 6];  
  
a = [a; x]                a =  
  
                                1  2  
                                3  4  
                                5  6
```

Subscripts

- Individual elements of a matrix are referenced with two subscripts, the first for the row the second for the column
- Less commonly, a single subscript can be used (column-based order)

```
>> CreateM
```

```
a1 = a(3,2);  
a2 = a(5);  
  
disp(['a(3,2) = ' num2str(a1)]);  
disp(['a(5) = ' num2str(a2)]);
```

```
a =  
1 2  
3 4  
5 6  
  
a(3,2) = 6  
a(5) = 4
```


Transpose

- The transpose operator turns rows into columns and vice-versa

a =

```
a = [1:3;4:6]  
b = a'
```

1	2	3
4	5	6

b =

1	4
2	5
3	6



The colon operator

- The colon operator is extremely powerful, and provides for very efficient ways of handling matrices

```
a =                                >> a(3,:)

1 2 3                                ans =
4 5 6
7 8 9                                7 8 9

>>
>> a(2:3,1:2)                         >> a(:,3)

ans =                                 ans =
                                         3
                                         6
                                         9
```

Tiling: Duplicating rows and columns

- Sometimes it is useful to generate a matrix where all the rows and columns are the same. **repmat** can be used.

```
>> Colon  
  
b =  
b = [1:3]                                ans =  
      1   2   3  
repmat(b, 3 , 1)                          1   2   3   1   2   3  
repmat(b, 3 , 2)                          1   2   3   1   2   3  
                                         ans =  
                                         1   2   3  
                                         1   2   3  
                                         1   2   3
```


Deleting rows and columns

- The colon operator and the empty array can be used to delete entire rows or columns

a =

1	2	3
4	5	6
7	8	9

```
a = [1:3; 4:6; 7:9]
a(:,2) = []
a(1,:) = []
```

a =

1	3
4	6
7	9

a =

4	6
7	9

Elementary matrices

- There are a group of functions to generate ‘elementary’ matrices that are used in a number of applications.
- With a single argument, they generate $n \times n$ square matrices, with two arguments they generate $n \times m$ matrices
- The function `eye(n)` generates an $n \times n$ identify matrix

```
>> ones(3)
```

ans =

```
1 1 1  
1 1 1  
1 1 1
```

```
>> zeros(2,3)
```

ans =

```
0 0 0  
0 0 0
```



MATLAB functions

- When a MATLAB mathematical or trigonometric function has a matrix argument, the function operates on every element of the matrix
- However, some MATLAB functions operate on matrices ***column by column***
- To test if all the elements of a matrix are true, two steps needed

```
a = [1 0 1; 1 1 1; 0 0 1];  
all(a)  
all(all(a))  
>> Functions  
ans =  
  
1x3 logical array  
  
0 0 1  
  
ans =  
  
logical  
  
0
```

Manipulating matrices

- Here are functions for manipulating matrices
 - `diag` extracts or creates a diagonal
 - `fliplr` flips from left to right
 - `flipud` flips from top to bottom
 - `rot90` rotates
 - `tril` extracts the lower triangular part

Element by element operations

- If a is a matrix $a * 2$ multiplies each element of a by 2, and $a . * 2$ also does

```
>> a
```

```
a =
```

```
1 0 1  
1 1 1  
0 0 1
```

```
>> 2*a
```

```
ans =
```

```
2 0 2  
2 2 2  
0 0 2
```

```
>> a
```

```
a =
```

```
1 0 1  
1 1 1  
0 0 1
```

```
>> 2 .* a
```

```
ans =
```

```
2 0 2  
2 2 2  
0 0 2
```



Matrix Operations: Multiplication

- Multiplication is probably the most important matrix operation
- Widely used in: network theory, solution of linear systems, population modeling
- AB is not equal to BA
- If A is an $n \times m$ matrix and B is an $m \times p$ matrix, their product C will be an $n \times p$ matrix, and the general element c_{ij} is given by:

$$c_{ij} = \sum_{k=1}^m a_{ik} b_{kj}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 & 6 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 5 & 4 \\ 15 & 14 \end{pmatrix}$$

```
a = [1 2; 3 4];  
b = [5 6; 0 -1];
```

```
c = a*b
```

```
c =
```

$$\begin{matrix} 5 & 4 \\ 15 & 14 \end{matrix}$$

Matrix Exponentiation

- A^2 means $A \times A$, where A must be a square matrix
- The operator \wedge is used for matrix exponentiation
- Note: why is $A \wedge 2$ different from $A .\wedge 2$?

```
a = [1 2; 3 4];
```

```
a ^ 2
```

```
ans =
```

```
7 10  
15 22
```



Other (more advanced) Matrix functions

- `det` – determinant
- `eig` – eigenvalue decomposition
- `inv` – inverse
- `expm` – exponential matrix
- `lu` – LU factorisation
- `qr` – orthogonal factorisation
- `svd` – singular value decomposition

Challenge 2.4

- Given the following matrices A and B, calculate results for the following operations in MATLAB, and explain the basis for your results.

$$A = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

A * B;

A .* B;

A + B;

A .^ B;



An Example

- A ready mix company has three factories (F1, F2 and F3) which must supply 3 building sites (S1, S2 and S2)
- The costs of transporting a load of concrete from any factory to any given site are given by the following table:

	S1	S2	S3
F1	3	12	10
F2	17	18	35
F3	7	10	24



Transportation Scheme

- Suppose the factory manager proposes the following transportation scheme (each entry represents the number of concrete loads to be transported along this particular route).

	S1	S2	S3
F1	4	0	0
F2	6	6	0
F3	0	3	5

Solution

- Each entry in the solution table must be multiplied by the corresponding element in the cost table.
- The values should then be summed
- Note, the solution is not the mathematical operation of matrix multiplication

	s1	s2	s3
F1	3	12	10
F2	17	18	35
F3	7	10	24

	s1	s2	s3
F1	4	0	0
F2	6	6	0
F3	0	3	5

MATLAB Code

```
C = [3 12 10; 17 18 35; 7 10 24];           >> Concrete
X = [4 0 0; 6 6 0; 0 3 5];
disp(C);
disp(X);
total = C .* X;
disp(total);
costs = sum(sum(total));
disp(['Total costs = '
num2str(costs)])
```

3 12 10	
17 18 35	
7 10 24	
4 0 0	
6 6 0	
0 3 5	
12 0 0	
102 108 0	
0 30 120	

Total costs = 372



CT248: Introduction to Modelling

3. Programming MATLAB Functions

Prof. Jim Duggan,
School of Computer Science
National University of Ireland Galway.
<https://github.com/JimDuggan/CT248>

Overview

- MATLAB allows you to create your own function M-files.
- A function differs from a script file in that the function M-file communicates with the MATLAB workspace only through specially designated input and output arguments
- Functions are indispensable tools when it comes to breaking a problem down into manageable pieces

MATLAB Function $y = f(x)$

- Write a function that takes in a vector and returns the mean and the standard deviation.

```
function [avg, stdev] = stats(x)
    avg = mean(x);
    stdev = std(x);
end
```

- Make use of the MATLAB functions `mean()` and `std()`

```
v = 1:5
v = 1 2 3 4 5
>> [a,s] = stats(v)
a = 3
s = 1.5811
```

General form of a function

- A function M-file *name.m* has the following general form

```
function[outarg1, outarg2, ...] = name(inarg1,...)
```

```
% comments to be displayed with help
```

```
...
```

```
outarg1 = ...;
```

```
outarg2 = ...;
```

```
...
```

Further information (1/5)

- **Function keyword**
 - The function file must start with the keyword function
- **Input and output arguments**
 - The input and output arguments define the function's means of communication with the workspace
- **Multiple output arguments**
 - If there is more than one output argument, the output arguments must be separated by commas and enclosed in square brackets
- Function names should follow the MATLAB rules for variable names
- **Help text**
 - When you type *help function_name*, MATLAB displays the comment lines that appear between the function definition line and the first non-comment line.

Further information (2/5)

- **Local variables: scope**
 - Any variables defined inside a function are accessible outside the function. These local variables exist only inside the function, which has its own workspace separate from the base workspace.
- **Global variables**
 - Variables which are defined in the base workspace are not normally accessible inside functions. If functions, and possibly the base workspace, declare variables as global, they all share single copies of those variables.

Challenge 3.1

Write a function **evens(v)** which returns the even values of a vector. A sample test run of the function is shown below.

```
>> v  
  
v =  
  
3   6   5   3   2   3   1   2   5   1  
  
>> v1 = evens(v)  
  
v1 =  
  
6   2   2
```



Challenge 3.2

- Write a function (`swap`) that takes:
 - A two dimensional array (m)
 - A target value (t)
 - A replacement value (r), and
- Replaces all occurrences of t in m with the value r

Global example

```
clear;

global X
X = 0;

function inc_X()
global X
X = X + 1;

inc_X();
inc_X();
inc_X();
inc_X();
inc_X();

disp('The value of X is...');

disp(X);
```

```
>> TestGlobal
The value of X is...
5
```

Further information (3/5)

- **Persistent variables**
 - A variable in a function may be declared persistent.
 - These (unlike local variables) remain persistent between function calls
 - A persistent variable is initially an empty array
- **Vector arguments**
 - Input and output variables can be vectors

Persistent example

```
function persist
persistent counter
    if isempty(counter)
        counter = 1;
    else
        counter = counter + 1;
    end
    disp(['The current counter = ' ...
        num2str(counter)]);

```

>> persist()
The current counter = 1

>> persist()
The current counter = 2

>> persist()
The current counter = 3

>> persist()
The current counter = 4

>> persist()
The current counter = 5

Further information (4/5)

- **How function arguments are passed**
 - If a function changes the value of any of its input arguments, the change is NOT reflected in the actual argument
 - An input argument is only passed by value if a function modifies it.
 - If a function does not modify an input argument, it is passed by reference.
- **Checking the number of function arguments**
 - A function may be called with all, some or none of its arguments. The same applies to output arguments
 - nargin displays the number of arguments passed
- A variable number of arguments can be passed.

Further information (5/5)

- Subfunctions
 - The function M-file may contain the code for more than one function. The first function is the primary function.
 - Additional functions are known as subfunctions, and are visible only to the primary function and the other subfunctions.
 - Subfunctions follow each other in any order AFTER the primary function.

Challenge 3.3

Write a function (m file) that processes elements of a 2-dimensional array on a row-by-row basis. The function should return 2 column vectors, the first containing the minimum value for each row, the second containing the maximum value of each row.

Furthermore, min and max *subfunctions* should be written to calculate the min and max of an individual row (i.e. the MATLAB min and max cannot be used).

Sample data for the problem (1 input and 2 outputs) is shown below.

$$\text{Input} = \begin{pmatrix} 10 & 20 \\ 50 & 40 \\ 80 & 60 \end{pmatrix} \quad \text{Min} = \begin{pmatrix} 10 \\ 40 \\ 60 \end{pmatrix} \quad \text{Max} = \begin{pmatrix} 20 \\ 50 \\ 80 \end{pmatrix}$$

CT248: Introduction to Modelling

4. Anonymous Functions in MATLAB

Prof. Jim Duggan,
School of Computer Science
National University of Ireland Galway.
<https://github.com/JimDuggan/CT248>

General form of a function

- A function M-file *name.m* has the following general form

```
function[outarg1, outarg2, ...] = name(inarg1,...)
```

% comments to be displayed with help

...

```
outarg1 = ...;
```

```
outarg2 = ...;
```

...



Function Handles

- A handle can be created to a function using @
`fhandle = @sqrt`
- Then the function feval can be used to activate the function through its handle
`feval(fhandle, 2)` or
`fhandle(2)`
- This is a very useful: we can now effectively pass functions to functions

Code Example

```
>> misc(@sqrt,100);

answ =

function [answ] = misc(fh, data)
answ = feval(fh, data)

>> misc(@sin,100);

answ =

-0.5064
```



Challenge 4.1

- Write a function that returns handles to two subfunctions
- These subfunctions are then called from the workspace
- The function is: `get_functions()`, and returns the handles to the sub-functions
- The subfunctions are:
 - `sum_rows(m)` – gets the sum of the rows in a matrix
 - `sum_cols(m)` – get the sum of the columns in a matrix
- Test the sub-functions using a separate script

Anonymous Functions

- Provide a quick means of creating simple functions without having to create M-files each time
- Can be constructed at the command line or in a script file
- Syntax:
`fhandle = @(arglist) expr`



`fhandle = @(arglist) expr`

- **expr** represents the body of the function, and consists of any single, valid MATLAB expression
- **arglist** is a comma-separated list of input arguments to be passed to the function.
- **@** constructs a function handle, and is required as part of an anonymous function definition.
- The LHS (function handle) can then be used just like any MATLAB variable

Example (1 parameter)

f =

```
f = @(x) x.*2  
f(1:10)
```

function_handle with value:

@(x)x.*2

ans =

2 4 6 8 10 12 14 16 18 20



Example (two parameters)

```
g = @(x,y) sum(x.*y)  
g([1 2 3], [4 5 6])
```

g =

function_handle with value:

@(x,y)sum(x.*y)

ans =

32



Example – returning a matrix

```
h = @(x) [x .* 2; x.^2]  
h(1:5)
```

h =

function_handle with value:

@(x)[x.*2;x.^2]

ans =

2 4 6 8 10
1 4 9 16 25



Challenge 4.2

- Write an anonymous function that returns the response to the equation of a line
- Inputs should be:
 - An x vectors of line points
 - The slope of the line (m)
 - The intercept (c)
- Plot the results on a graph



Challenge 4.2

- Write an anonymous function that takes in a vector and returns all the odd numbers in a new vector



Challenge 4.3

- Explain what is happening in the following four lines of MATLAB code, and show what the values of y1 and y2 will be.

```
f1 = @max  
f2 = @min  
y1 = feval(f1,10,20);  
y2 = feval(f2, 10, 20);
```

- What are the potential benefits of using @ and feval, and name a MATLAB function that makes use of these mechanisms.

Challenge 4.5

- Explain what is happening in the following MATLAB code, and determine the values (and type) of the output.

```
f = @(x) [sum(x); min(x); max(x)]
```

```
f(1:5)
```



Challenge 4.4

Write a function (m file) that takes a 2-dimensional array and an input number. It should then create an output 2-dimensional array that contains only those values of the 2-dimensional array that are greater than the input number. For example, if input number is 5, and the input array (A) is

```
A =  
1 2 3  
4 5 6  
7 8 9
```

Then the function output should be.

```
ans =  
0 0 0  
0 5 6  
7 8 9
```



CT248: Introduction to Modelling

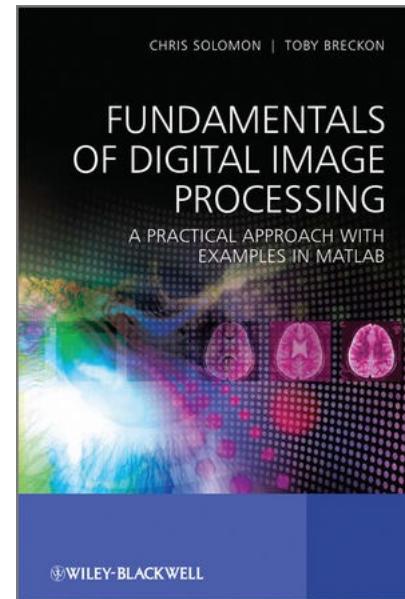
5. Processing Images in MATLAB

Prof. Jim Duggan,
School of Computer Science
National University of Ireland Galway.

<https://github.com/JimDuggan/CT248>

Overview

- Introduction to image processing with MATLAB (matrices and arrays)
- Image types
 - Binary
 - Grayscale
 - RGB
- Transformations on images



"A digital image can be considered as a discrete representation of data possessing both spatial (layout) and intensity (colour) information"

Image Types – Some Examples

Image Type	Description
Binary	2-D arrays that assign one numerical value from the set {0,1} to each pixel in the image. Often called logical images. Black corresponds to zero, white to a one. Can be represented as a simple bit stream.
Intensity/ Greyscale	2-D arrays that assign one numerical value to each pixel which is representative of the intensity at this point. Range is bounded by the bit resolution of the image.
RGB	3-D arrays that assign three numerical values to each pixel, each value corresponding to the red, green and blue (RGB) image channel. Pixels accessed in MATLAB by $I(\text{Row}, \text{Column}, \text{channel})$

Key MATLAB Functions

- **imread(filename)**
 - Reads an image from a graphics file, and converts to a MATLAB array object
 - <https://uk.mathworks.com/help/matlab/ref/imread.html>
- **imshow(object)**
 - Displays an image
 - <https://uk.mathworks.com/help/images/ref imshow.html>
- Note, MATLAB's image processing toolbox contains an extensive image processing library (manipulation & feature extraction).

```
I=imread('cameraman.tif');
```

```
imshow(I);
```



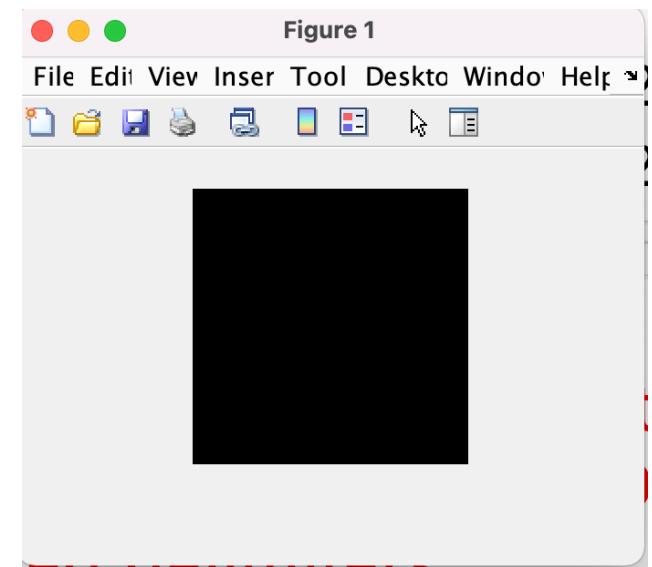
```
>> whos I
Name      Size            Bytes  Class    Attributes
I         256x256        65536  uint8
```

Binary Image - Example

Image Type	Description
Binary	2-D arrays that are assigned one numerical value from the set {0,1} to each pixel in the image. Often called logical images. Black corresponds to zero, white to a one. Can be represented as a simple bit stream.

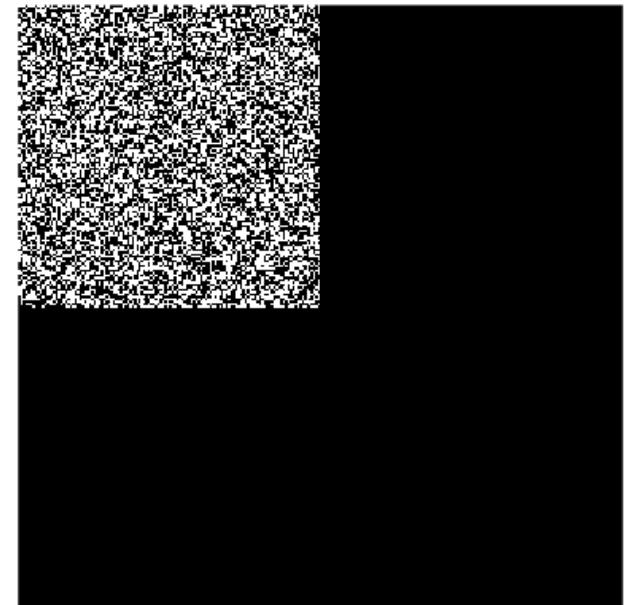
```
1 clear;
2
3 x1 = zeros(256,256);
4 imshow(x1);

>> whos
Name      Size            Bytes  Class     Attributes
x1    256x256           524288  double
```

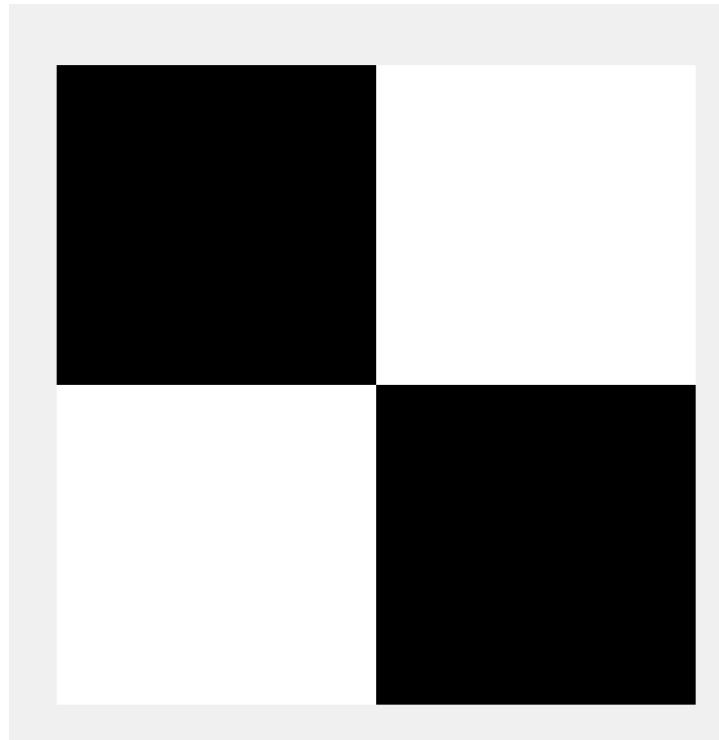


Add random white elements...

```
6     rng(100);  
7     x2 = x1;  
8     x2(1:(256/2),1:(256/2)) = ...  
9         randi([0 1], 256/2, 256/2);
```



Challenge 5.1 Create the following binary image (256x256)



Greyscale

Image Type	Description
Intensity/ Greyscale	2-D arrays that assign one numerical value to each pixel which is representative of the intensity at this point. Range is bounded by the bit resolution of the image. Range can be 0-255, unsigned integers.

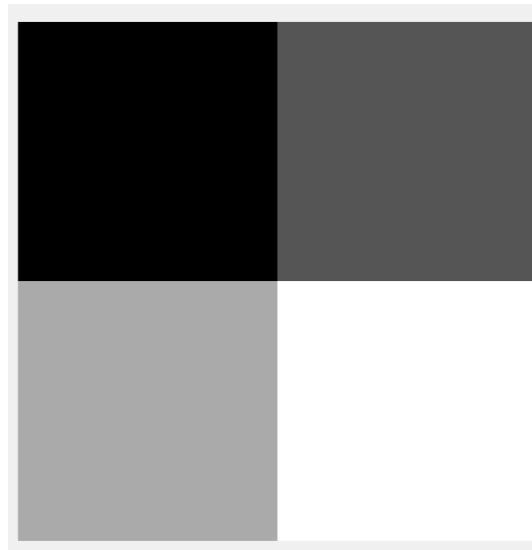


Image Colour

- An image contains one of more colour channels that define the intensity or colour at a particular pixel location $I(m,n)$
- In the simplest case, each pixel location only contains a single numerical value representing the signal level at that point in the image.
- Most common colour map is greyscale.
- Max number is $2^8 - 1 = 255$



```
>> whos I
  Name      Size            Bytes  Class    Attributes
  I         256x256        65536  uint8
>> I(1:3,1:3)
  3x3 uint8 matrix
  156  159  158
  160  154  157
  156  159  158
```

`linspace()` – A useful function

```
>> help linspace
```

linspace Linearly spaced vector.

linspace(X1, X2) generates a row vector of 100 linearly equally spaced points between X1 and X2.

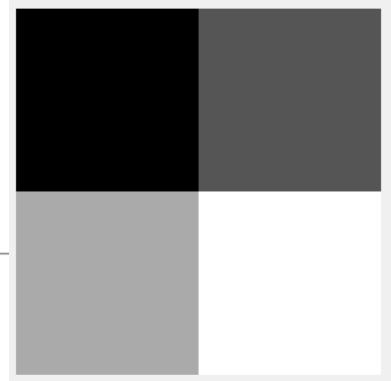
linspace(X1, X2, N) generates N points between X1 and X2.
For N = 1, **linspace** returns X2.

```
>> linspace(0,255,4)
```

ans =

```
0 85 170 255
```

Create the grid.



```
1 clear;
2
3 locs = linspace(0,255,4);
4 sq1 = uint8(locs(1) * ones(64,64));
5 sq2 = uint8(locs(2) * ones(64,64));
6 sq3 = uint8(locs(3) * ones(64,64));
7 sq4 = uint8(locs(4) * ones(64,64));
8
9 sq = [sq1 sq2; sq3 sq4];
10
11 imshow(sq);
```

Operations on Pixels

- The most basic type of image processing is a point transform which maps the values at individual points (pixels) in the input image to corresponding points (pixels) in an output image
- In a mathematical sense, it's a one-to-one functional mapping from input to output.
- Types of operations:
 - Pixel Addition and Subtraction
 - Thresholding
 - RGB to Greyscale
 - Rotation (90°)
 - Simple Cropping

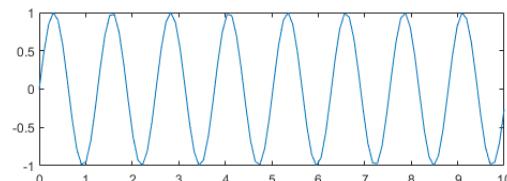
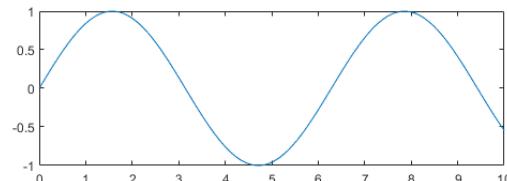


subplot() function

- Create axes in tiled positions
- subplot(m,n,p) divides the current figure into an m-by-n grid and creates axes in the position specified by p.
- MATLAB® numbers subplot positions by row.
- The first subplot is the first column of the first row, the second subplot is the second column of the first row, and so on.

```
subplot(2,1,1);  
x = linspace(0,10);  
y1 = sin(x);  
plot(x,y1);
```

```
subplot(2,1,2);  
y2 = sin(5*x);  
plot(x,y2)
```



Note, MATLAB operations on uint types

```
>> x = uint8(10);
>> whos
  Name      Size            Bytes  Class    Attributes
  x            1x1              1  uint8
>> x+246
ans =
  uint8
  255
```



Arithmetic Operations on images

Basic arithmetic operations can be performed quickly and easily on image pixels (contrast adjustment)

```
I = imread('cameraman.tif');
```

```
O = I + 50;  
O1 = I + 100;  
O2 = I - 100;
```

```
subplot(2,2,1),imshow(I),title('Original Image');  
subplot(2,2,2),imshow(O),title('+50');  
subplot(2,2,3),imshow(O1),title('+100');  
subplot(2,2,4),imshow(O2),title('-100');
```



Challenge 5.2

- Write an anonymous function called flip() that uses the MATLAB function rot90() to flip a matrix. Explore rot90() on a sample matrix. The following output should be generated.



Thresholding

- Produces a binary image from a greyscale or colour image by setting pixels to 1 or 0 depending on whether they are above or below the threshold value
- Logical operators very useful for this (TRUE = White, FALSE= Black).
- Useful to help separate the image foreground from background

A =

1	2	3
4	5	6
7	8	9

>>

>> A > 5

ans =

3x3 logical array

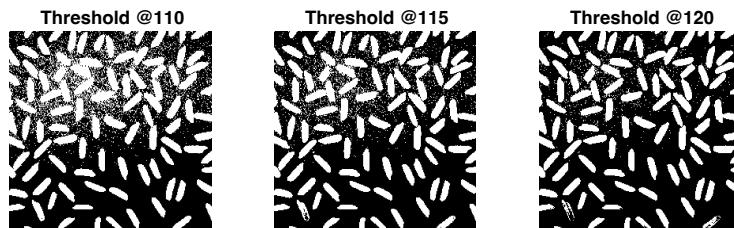
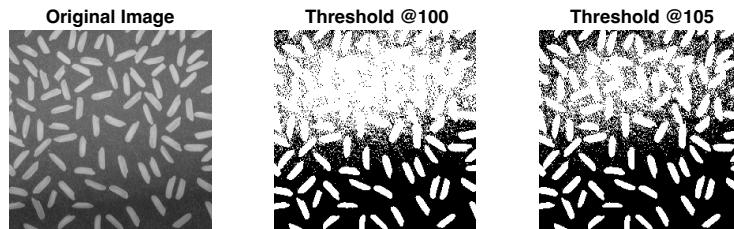
0	0	0
0	0	1
1	1	1

Threshold Example

```
I = imread('rice.png');
```

```
T1 = I > 100;  
T2 = I > 105;  
T3 = I > 110;  
T4 = I > 115;  
T5 = I > 120;
```

```
subplot(2,3,1),imshow(I),title('Original Image');  
subplot(2,3,2),imshow(T1),title('Threshold  
@100');  
subplot(2,3,3),imshow(T2),title('Threshold  
@105');  
subplot(2,3,4),imshow(T3),title('Threshold  
@110');  
subplot(2,3,5),imshow(T4),title('Threshold  
@115');  
subplot(2,3,6),imshow(T5),title('Threshold  
@120');
```

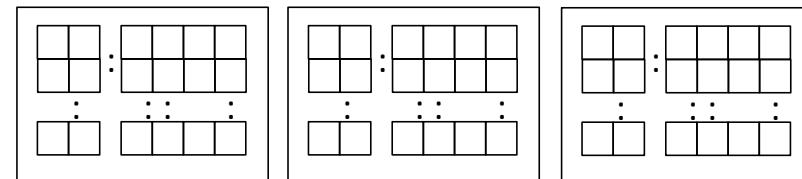


RGB Images

Image Type	Description
RGB	3-D arrays that assign three numerical values to each pixel, each value corresponding to the red, green and blue (RGB) image channel. Pixels accessed in MATLAB by $I(\text{Row}, \text{Column}, \text{channel})$



```
>> whos D
Name      Size            Bytes Class Attributes
D         135x198x3        80190 uint8
```



Colour Information

2.1 Web Colour Palette

PRIMARY COLOURS	#05A7AB	#9CCF00	#E16D2D	#49B0C6
	#027F8E	#7DB900	#CF652A	#015788

Primary colours are used throughout the site design.

TEAL is used as the base colour, appearing in the navigation, text headers and links.

LIME GREEN is the primary action item colour, appearing as the Apply Now button.

TERRACOTTA is used in events notifications on the inner pages.

SKY BLUE is used on the News section of the homepage and on secondary buttons.

DARK BLUE is featured in the Utility Navigation and Footer.

ACCENT COLOURS	#A34A88	#F7C600	#B2D9E0	#B5BFBF
	#8E3874	#F5B10D	#97C2CA	#5E6464
	#7D306F		#4E5153	#414545

Accent colours are used sparingly on the website.

MULBERRY PURPLES are used on the feature roundal ribbons, on inner pages.

YELLOW is featured in the Desktop Utility Navigation, as the 'High Contrast Text' option.

LIGHT BLUE is used to indicate the Downloadable Content on inner pages.

LIGHT GREY is used on the homepage icons.

DARK GREY is featured in the body copy.

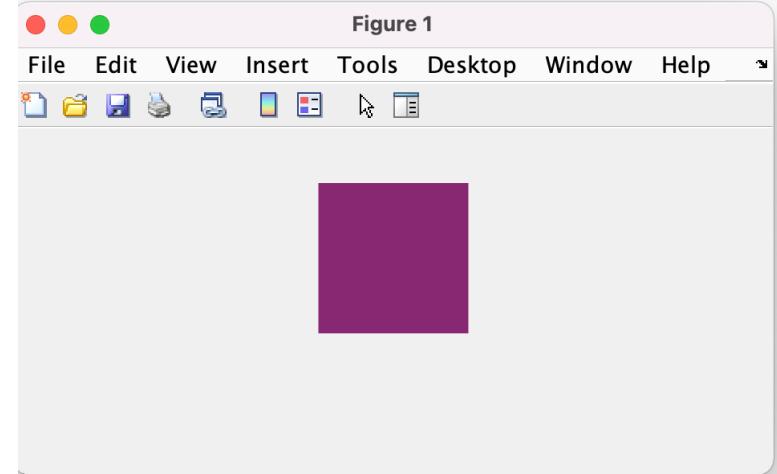
CHARCOAL GREY is featured in the footer, dropdown menus and mobile navigation.

2.2 Web Fonts

Creating a colour

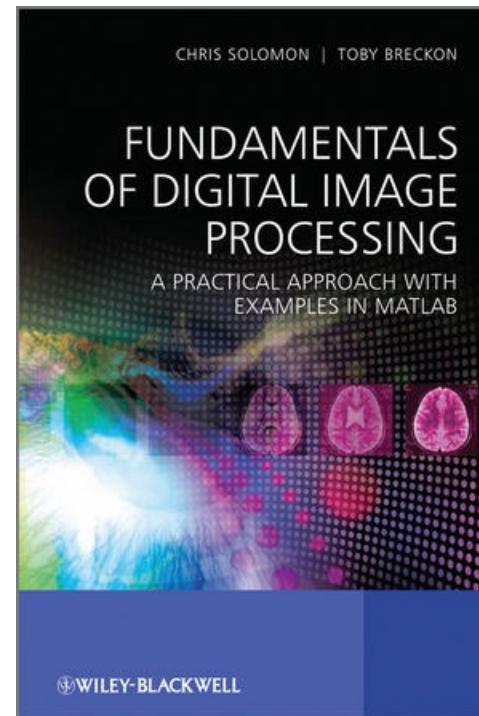


```
1 r_Str    = '7D'; g_Str    = '30'; b_Str    = '6F';
2
3 R = hex2dec(r_Str); G = hex2dec(g_Str);
4 B = hex2dec(b_Str);
5
6 pic = uint8(zeros(100,100,3));
7
8 pic(:,:,1) = R;
9 pic(:,:,2) = G;
10 pic(:,:,3) = B;
11
12 imshow(pic);
```



Summary

- Introduction to Image processing
- 3 Image types (binary, greyscale, RGB)
- MATLAB array processing very powerful
- Further topics:
 - Enhancement
 - Frequency domain processing
 - Image restoration
 - Geometry
 - Morphological Processing
 - Features
 - Image Segmentation
 - Classification



CT248: Introduction to Modelling

7. Introduction to Modelling

Prof. Jim Duggan,

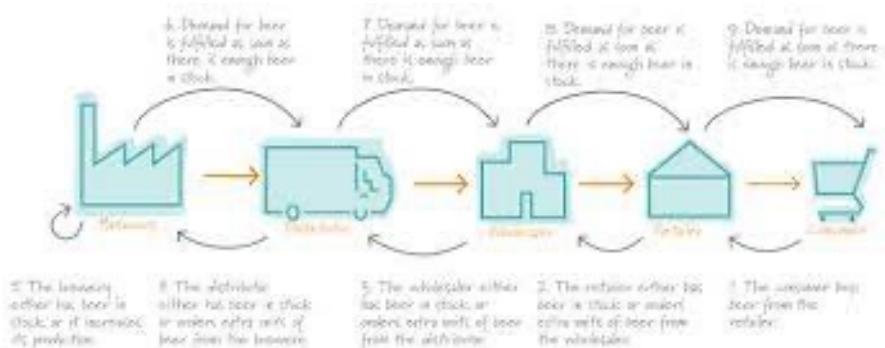
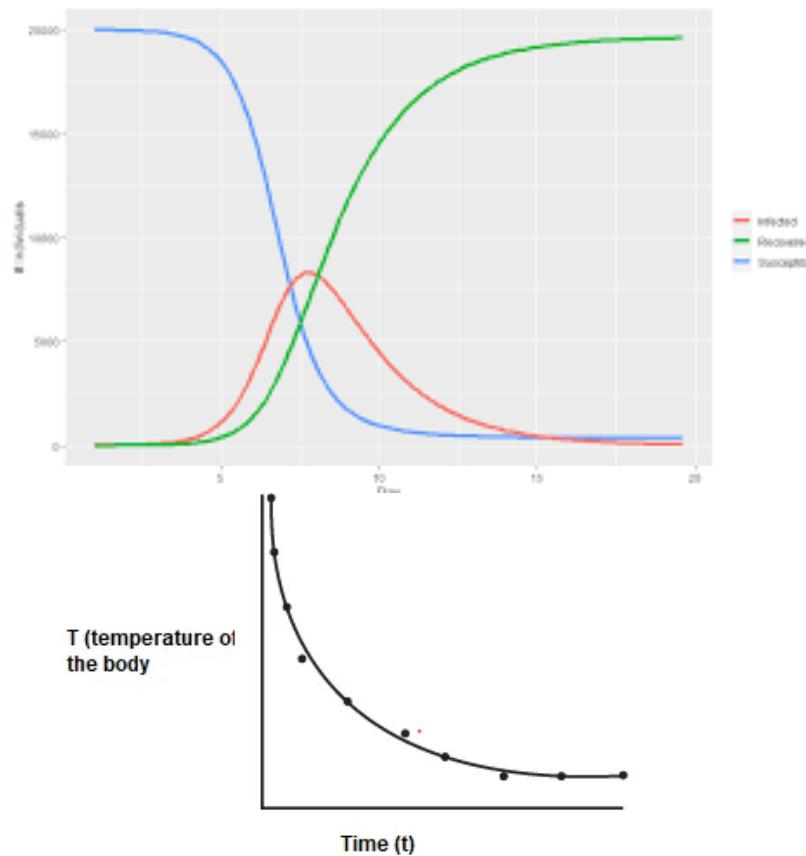
School of Computer Science

National University of Ireland Galway.

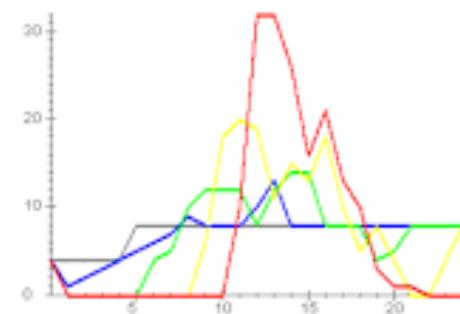
<https://github.com/JimDuggan/CT248>



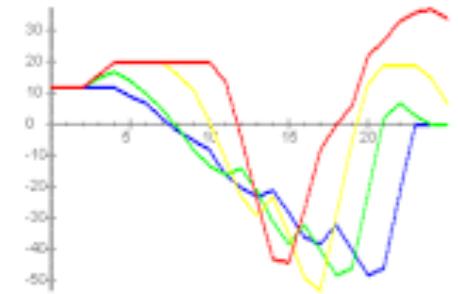
Computational Modelling – *Predicting variables over time*



Orders



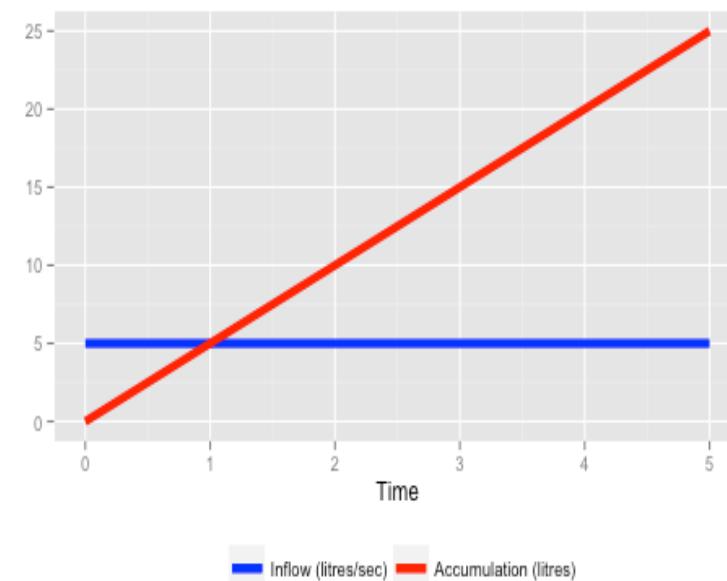
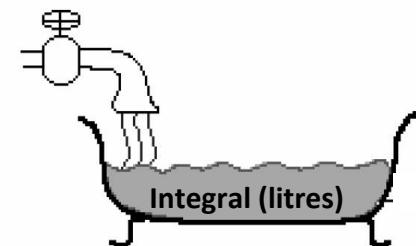
Stocks



Modelling Foundation - Integration

- Calculus is the study of how things **change over time**, and is described by Strogratz (2009) as “*perhaps the greatest idea that humanity has ever had.*”
- Integration is the mathematical process of calculating the area under the net flow curve, between initial and final times.
- Given the flows and the initial condition, integration provides the stock values (how much is in the bathtub at any future time t).

Derivative (litres/minute)



Integration

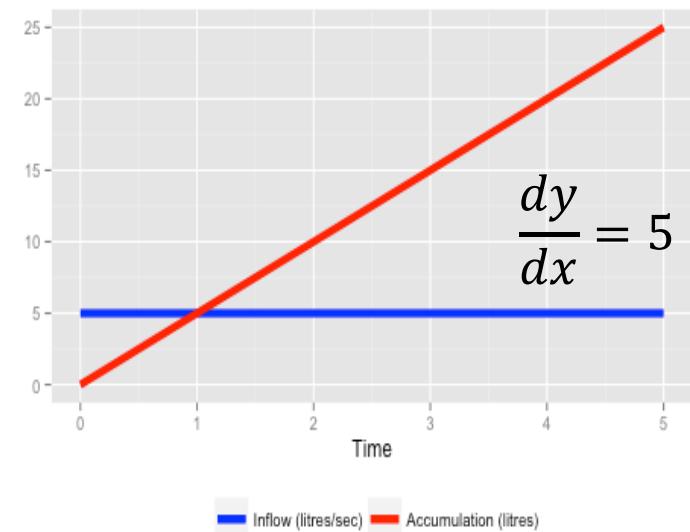
$$\int x^n \, dx = \frac{1}{n+1} x^{n+1} + c$$

$$f(x) = 5x^0$$

$$\int 5x^0 \, dx = 5 \int x^0 \, dx = \boxed{5x^1 + c}$$

$$\int_0^5 5x^0 \, dx = 5(5) - 5(0) = 25$$

$$\int_0^{1000} 5x^0 \, dx = 1000(5) - 5(0) = 5,000$$

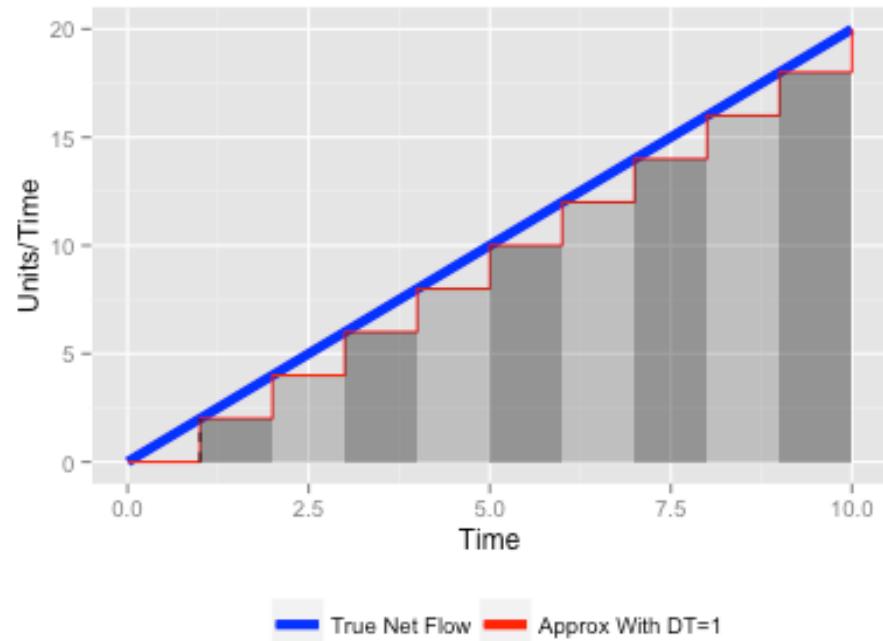


Field	Integral	Derivative
Mathematics, physics and engineering	Integrals, states, state variables, stocks	Derivatives, rates of change, flows
Chemistry	Reactants and reaction products	Reaction rates
Manufacturing	Buffers, inventories	Throughput
Economics	Levels	Rates
Accounting	Stocks, balance sheet items	Flows, cash flow or income statement items
Biology, physiology	Compartments	Diffusion rates, flows
Medicine, epidemiology	Prevalence, reservoirs	Incidence, infection, morbidity and mortality rates



Numerical Integration

- Euler's Method (Integration Algorithm)
- Approximate area under the net flow curve as a summation of rectangles, of width DT
- The smaller DT, the more accurate the result



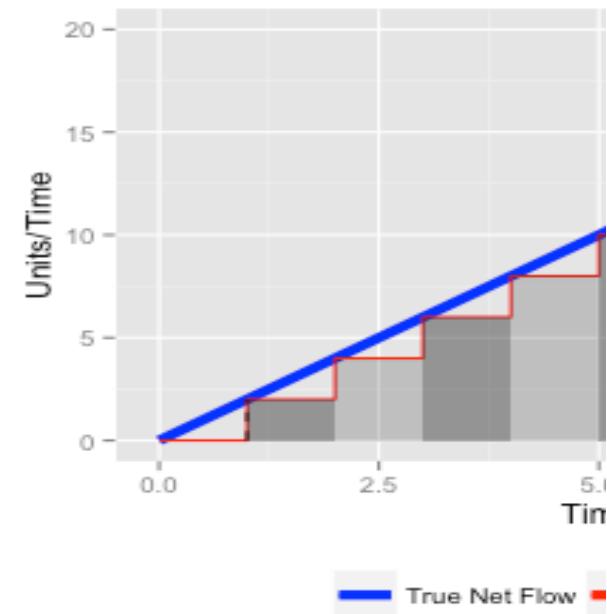
$$S_t = S_{t-dt} + NF_{t-dt} \times DT$$

$$\frac{dy}{dt} = 2t$$

Numerical Integration

$$S_t = S_{t-dt} + NF_{t-dt} \times DT$$

t	Y	NF = 2 * t	Y Exact (t ²)	Diff (Y, Y Exact)
0	0	0	0	0
1	0	2	1	1
2	2	4	4	2
3	6	6	9	3
4	12	8	16	4
5	20	10	25	5



MATLAB Provides Functions

[t,y] = ode45(odefun,tspan,y0), where tspan = [t0 tf]

integrates the system of differential equations $y' = f(t,y)$ from t0 to tf with initial conditions y0.

Each row in the solution array y corresponds to a value returned in column vector t.

An anonymous function can be used for odefun. It should return a column vector of solutions

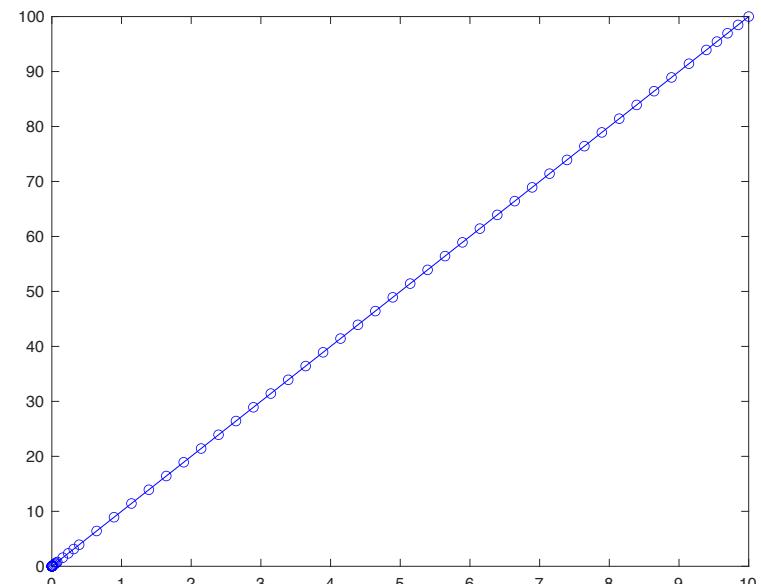
Example 1 – Rate of Change is constant

$$\frac{dy}{dt} = 10$$

```
function y = model_01(t, x)
y = 10;
end

[t,y] = ode45(@model_01, [0 10], 0);
plot(t, y, '-ob');

>> whos
Name      Size            Bytes  Class    Attributes
t         69x1             552  double
y         69x1             552  double
```



Code Summary

Model Function definition

```
function y = model_01(t, x)
y = 10;
end
```

MATLAB's ode solver

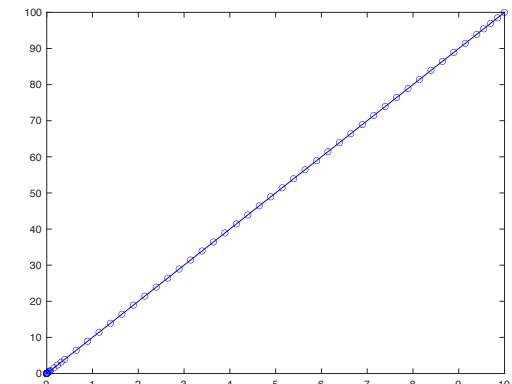
Time vector

```
[t, y] = ode45(@model_01, [0 10], 0);
```

Initial conditions ($Y = 0$)

Function handle that solves equation(s)

Returns time vector and solution vector into two equal-size vectors. (y could also be an array).



Exploring the solver ode45()

```
function y = model_01(t, x)
persistent counter;
if isempty(counter)
counter = 1;
else
counter = counter + 1;
end

fprintf("Calling model_01 T=%f x=%f
Counter=%d\n",t,x,counter);
y = 10;
end
```

```
>> run_model_01
Calling model_01 T=0.000000 x=0.000000 Counter=1
Calling model_01 T=0.000004 x=0.000040 Counter=2
Calling model_01 T=0.000006 x=0.000060 Counter=3
Calling model_01 T=0.000016 x=0.000161 Counter=4
Calling model_01 T=0.000018 x=0.000179 Counter=5
Calling model_01 T=0.000020 x=0.000201 Counter=6
Calling model_01 T=0.000020 x=0.000201 Counter=7

Calling model_01 T=9.281366 x=92.813661 Counter=95
Calling model_01 T=9.392477 x=93.924772 Counter=96
Calling model_01 T=9.392477 x=93.924772 Counter=97
Calling model_01 T=9.513982 x=95.139818 Counter=98
Calling model_01 T=9.574734 x=95.747341 Counter=99
Calling model_01 T=9.878495 x=98.784954 Counter=100
Calling model_01 T=9.932497 x=99.324975 Counter=101
Calling model_01 T=10.000000 x=100.000000 Counter=102
Calling model_01 T=10.000000 x=100.000000 Counter=103
```



Getting equi-distant outputs

```
[t,y] = ode45(@model_01, [0:10], 0);
plot(t, y, '-ob');

>> whos
  Name      Size            Bytes  Class     Attributes
  t         11x1             88  double
  y         11x1             88  double

>> [t y]
ans =
          0      0
    1.0000  10.0000
    2.0000  20.0000
    3.0000  30.0000
    4.0000  40.0000
    5.0000  50.0000
    6.0000  60.0000
    7.0000  70.0000
    8.0000  80.0000
    9.0000  90.0000
   10.0000 100.0000
```



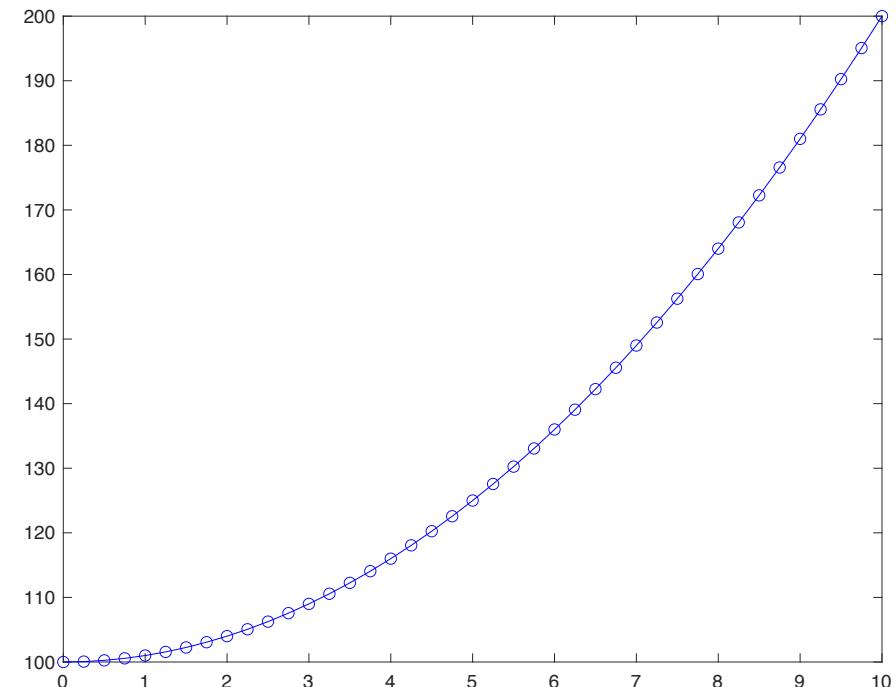
Example 2 – Time varying rate of change

$$\frac{dy}{dt} = 2t$$

```
function y = model_02(t, x)
y = 2*t;
end
```

Note: Now using an input from the solver ($t = \text{time}$)

```
[t,y] = ode45(@model_02, [0 10], 100);
plot(t, y, '-ob');
```



Example 3 – Two equations

$$\frac{dy}{dt} = 2t$$

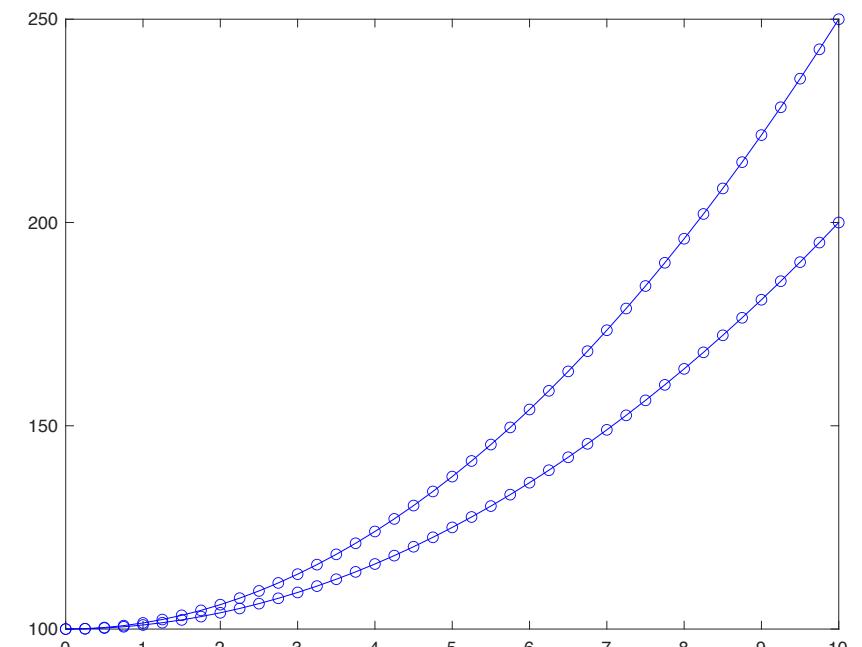
$$\frac{dz}{dt} = 3t$$

```
function y = model_03(t, x)
y = [0;0];
y(1) = 2*t;
y(2) = 3*t;
end
```

Note: y is now a column vector. ode45 requires this format.

```
[t,y] = ode45(@model_03, [0 10], [100 100]);
plot(t, y, '-ob');
```

Note: a vector of initial conditions is now required. The size must align with the size of y in the model function.



Example 4 – Exponential growth (Population)

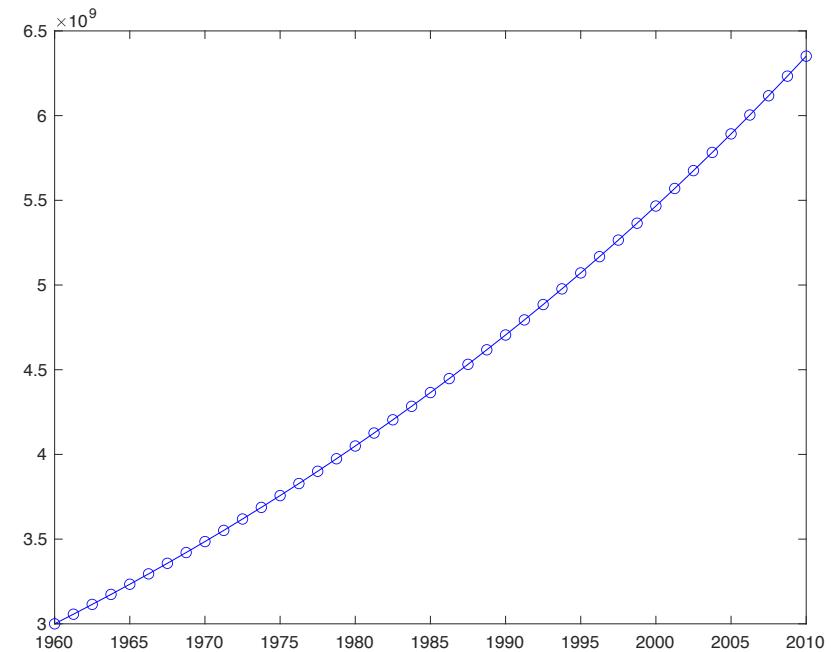
$$\frac{dP}{dt} = rP$$

```
function y = model_04(t, x, r)
y = r*x;
end
```

Note: (1) We've added a new parameter r . This will be passed in via the call to `ode45()`. (2) The variable x is used for the first time. This allows us to use the value of P in the equation.

```
[t, y] = ode45(@model_04, [1960 2010],
3e9,odeset,0.015);
plot(t, y, '-ob');
```

Note: The value that will be copied to r is the 5th parameter.
We need to type `odeset` as the default 4th parameter.

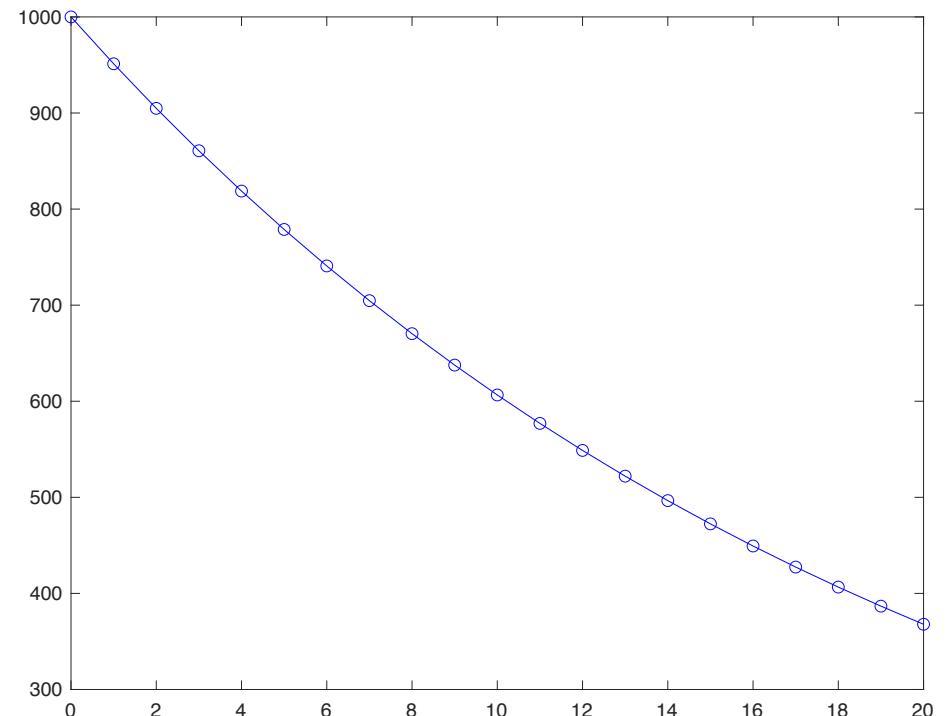


Example 5 – Exponential decline

$$\frac{dP}{dt} = -rP$$

```
function y = model_04(t, x, r)
y = -r*x;
end
```

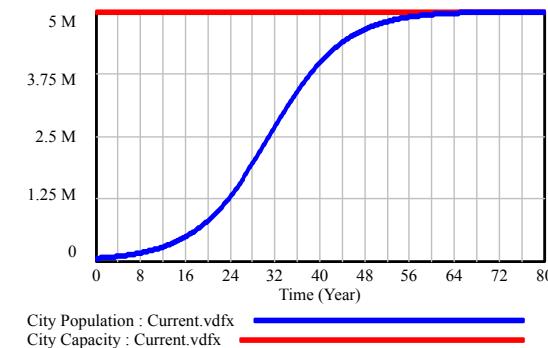
```
[t,y] = ode45(@model_04, [0:20], 1000,
odeset,0.05);
plot(t, y, '-ob');
```



Example 6 - Verhulst Model



- A model of population growth, where the rate of increase is limited by the carrying capacity (K)
- When P is small, it approximates exponential growth
- Model compared to population growth in:
 - France (1817-1831)
 - Belgium (1815-1833)
 - Essex (1811-1831)



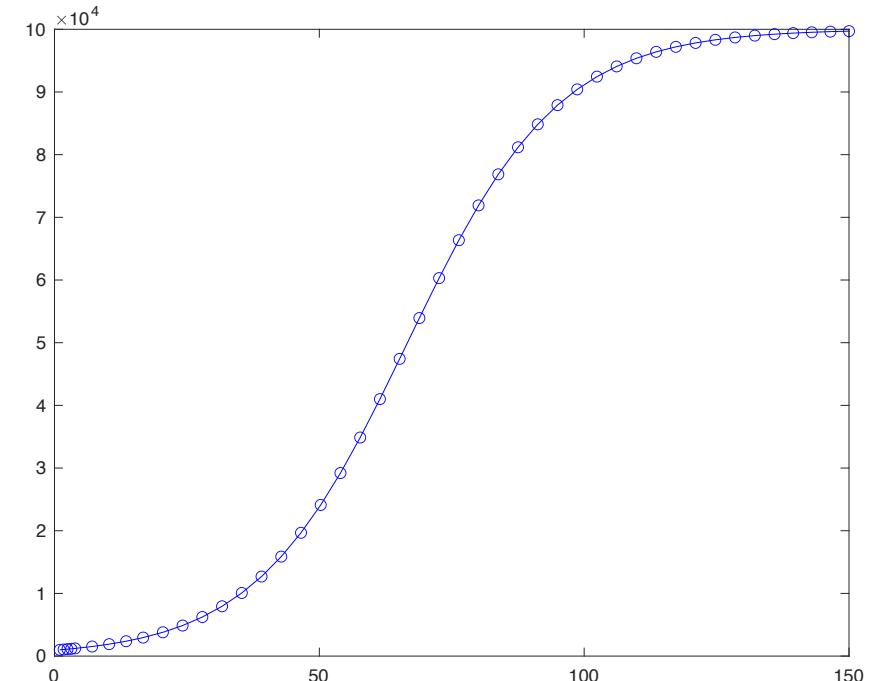
$$\frac{dP}{dt} = r P \left(1 - \frac{P}{K}\right)$$

Example 6 - Verhulst Model

$$\frac{dP}{dt} = r P \left(1 - \frac{P}{K}\right)$$

```
function y = model_06(t, x, r, K)
y=r*x*(1-x/K);
end

[t,y] = ode45(@model_06, [1 150], 1000,
odeset,0.07,100000);
plot(t, y, '-ob');
```



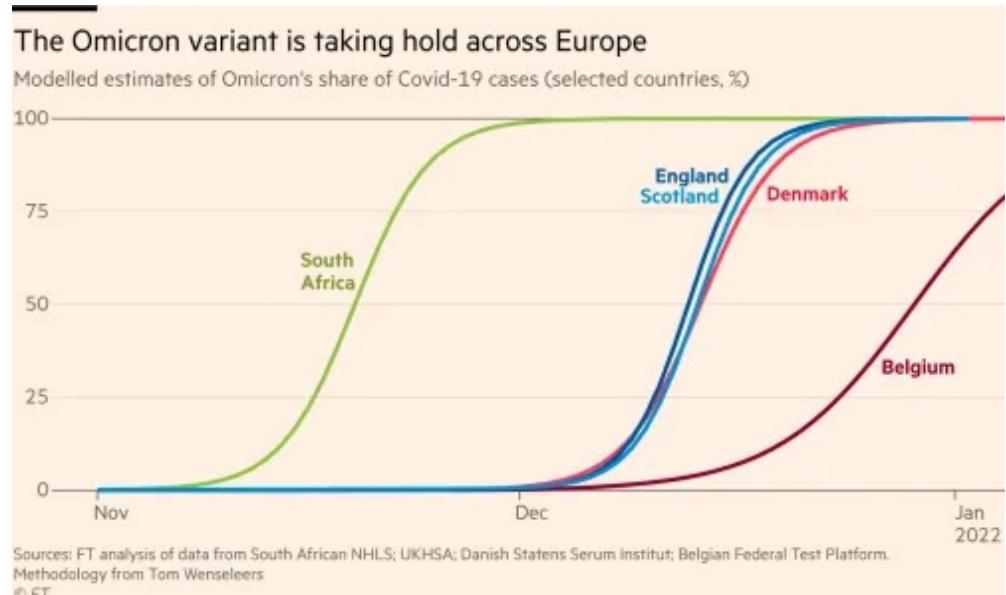
Challenge 6.1

- Implement all the six models as anonymous functions



Challenge 6.2

- Explore how the Verhulst model might be used to model the growth of a virus variant. What values might be used for
 - Growth rate
 - Initial virus level
 - Maximum virus level



<https://www.ft.com/content/3c27c135-fdbc-4db7-8c7c-6e1f6c386235>

CT248: Introduction to Modelling

8. Exploring Data with MATLAB

Prof. Jim Duggan,

School of Computer Science

National University of Ireland Galway.

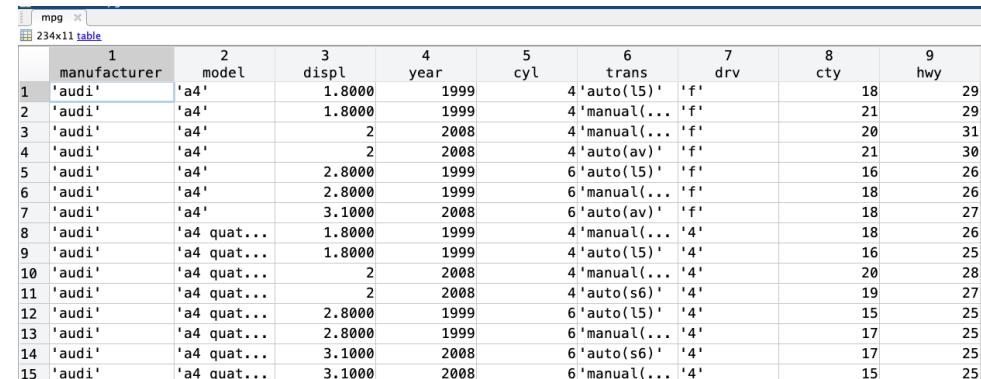
<https://github.com/JimDuggan/CT248>

Exploratory Data Analysis

- An iterative cycle
 - Generate questions about your data
 - Search for answers by visualizing, transforming, and modelling your data
 - Use what you learn to refine your questions and/or generate new questions

```
mpg = readtable("mpg.xlsx");
```

```
>> whos
Name      Size      Bytes Class Attributes
mpg      234x11    174280  table
```



The screenshot shows the MATLAB workspace with a table named 'mpg'. The table has 234 rows and 11 columns. The columns are labeled 1 through 11. Column 1 is 'manufacturer', column 2 is 'model', column 3 is 'displ', column 4 is 'year', column 5 is 'cyl', column 6 is 'trans', column 7 is 'drv', column 8 is 'cty', and column 9 is 'hwy'. The 'trans' and 'drv' columns contain categorical values like 'auto(l5)', 'manual', etc. The 'cyl' column contains numerical values ranging from 4 to 8. The 'year' column contains years from 1999 to 2008. The 'displ' column contains engine displacement values.

	1	2	3	4	5	6	7	8	9	
	manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	
1	'audi'	'a4'	1.8000	1999	4	'auto(l5)'	'f'	18	29	
2	'audi'	'a4'	1.8000	1999	4	'manual(...'	'f'	21	29	
3	'audi'	'a4'	2	2008	4	'manual(...'	'f'	20	31	
4	'audi'	'a4'	2	2008	4	'auto(av)'	'f'	21	30	
5	'audi'	'a4'	2.8000	1999	6	'auto(l5)'	'f'	16	26	
6	'audi'	'a4'	2.8000	1999	6	'manual(...'	'f'	18	26	
7	'audi'	'a4'	3.1000	2008	6	'auto(av)'	'f'	18	27	
8	'audi'	'a4 quatt...	1.8000	1999	4	'manual(...'	'4'	18	26	
9	'audi'	'a4 quatt...	1.8000	1999	4	'auto(l5)'	'4'	16	25	
10	'audi'	'a4 quatt...	2	2008	4	'manual(...'	'4'	20	28	
11	'audi'	'a4 quatt...	2	2008	4	'auto(s6)'	'4'	19	27	
12	'audi'	'a4 quatt...	2.8000	1999	6	'auto(l5)'	'4'	15	25	
13	'audi'	'a4 quatt...	2.8000	1999	6	'manual(...'	'4'	17	25	
14	'audi'	'a4 quatt...	3.1000	2008	6	'auto(s6)'	'4'	17	25	
15	'audi'	'a4 quatt...	3.1000	2008	6	'manual(...'	'4'	15	25	

Defining Important Terms (Wickham and Grolemund)

Term	Explanation
Variable	A quantity, quality or property that you can measure
Value	The state of a variable when you measure it. The value of a variable may change from measurement to measurement
Observation (or case)	A set of measurements made under similar conditions (you usually make all of the measurements in an observation at the same time and on the same object). An observation will contain several values, each associated with a different variable.
Tabular data	A set of values, each associated with a variable and an observation. Tabular data is tidy if each value is placed in its own “cell”, each variable in its own column, and each observation in its own row.

Fuel Economy Data Set (ggplot2::mpg)

This dataset contains a subset of the fuel economy data that the EPA makes available on <http://fueleconomy.gov>. It contains only models which had a new release every year between 1999 and 2008 - this was used as a proxy for the popularity of the car.

manufacturer	manufacturer	drv	f = front-wheel drive, r = rear wheel drive, 4 = 4wd
model	model name	cty	city miles per gallon
displ	engine displacement, in litres	hwy	highway miles per gallon
year	year of manufacture	fl	fuel type
cyl	number of cylinders	class	“type” of car
trans	type of transmission		

Introducing the table in MATLAB

- Tables are used to collect heterogeneous data and metadata into a single container.
- Tables are suitable for storing column-oriented or tabular data that are often stored as columns in a text file or in a spreadsheet.
- Tables can accommodate variables of different types, sizes, units, etc.
- They are often used to store experimental data, with rows representing different observations and columns representing different measured variables.

Tables...

- Tables can be subscripted using parentheses much like ordinary numeric arrays, but in addition to numeric and logical indices, you can use a table's variable and row names (if defined) as indices.
- You can access individual variables in a table much like fields in a structure, **using dot subscripting**.
- You can access the contents of one or more variables using brace subscripting.

Example

```
clear;  
  
mpg = readtable("mpg.xlsx");  
  
mpg(1:2,1:5)  
  
mpg.cty(1:3)  
  
mpg(1:3, {'cty' 'hwy'})
```

ans =

2×5 [table](#)

manufacturer	model	displ	year	cyl
{'audi'}	{'a4'}	1.8	1999	4
{'audi'}	{'a4'}	1.8	1999	4

ans =

ans =

3×2 [table](#)

cty	hwy
18	29
21	29
20	31



Cell Array

- A *cell array* is a data type with indexed data containers called *cells*, where each cell can contain any type of data.
- Cell arrays commonly contain either lists of text, combinations of text and numbers, or numeric arrays of different sizes.
- Refer to sets of cells by enclosing indices in smooth parentheses, () .
- Access the contents of cells by indexing with curly braces, {}.

Code Example

```
c = {42, rand(5), "abcd"};                                >> c(2)  
ans =  
  
>> c(1)                                              1x1 cell array  
  
ans =                                              {5x5 double}  
  
1x1 cell array                                >>  
{[42]}                                         >> c{2}  
  
>> c{1}                                              ans =  
  
ans =  
42                                                 0.8147  0.0975  0.1576  0.1419  0.6557  
0.9058  0.2785  0.9706  0.4218  0.0357  
0.1270  0.5469  0.9572  0.9157  0.8491  
0.9134  0.9575  0.4854  0.7922  0.9340  
0.6324  0.9649  0.8003  0.9595  0.6787
```



Converting a cell to a string (1)

```
>> mpg(1:2,:)
```

ans =

2×11 table

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
{'audi'}	{'a4'}	1.8	1999	4	{'auto(l5)' }	{'f'}	18	29	{'p'}	{'compact'}
{'audi'}	{'a4'}	1.8	1999	4	{'manual(m5)'}	{'f'}	21	29	{'p'}	{'compact'}



Converting a cell to a string (2)

```
mpg.manufacturer = string(mpg.manufacturer);
```

```
>> mpg(1:2,:)
```

```
ans =
```

```
2×11 table
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
"audi"	{'a4'}	1.8	1999	4	{"auto(l5)" }	{'f'}	18	29	{'p'}	{"compact"}
"audi"	{'a4'}	1.8	1999	4	{"manual(m5)"}	{'f'}	21	29	{'p'}	{"compact"}

summary () function

```
>> summary(mpg)
```

Variables:

manufacturer: 234×1 cell array of character vectors

model: 234×1 cell array of character vectors

displ: 234×1 double

Values:

Min	1.6
Median	3.3
Max	7

year: 234×1 double

Values:

Min	1999
Median	2003.5
Max	2008

cyl: 234×1 double

Values:

Min	4
Median	6
Max	8

trans: 234×1 cell array of character vectors

drv: 234×1 cell array of character vectors

cty: 234×1 double

Values:

Min	9
Median	17
Max	35

hwy: 234×1 double

Values:

Min	12
Median	24
Max	44

fl: 234×1 cell array of character vectors

class: 234×1 cell array of character vectors



Additional Example

First, create workspace variables that have the patient data. The variables can have any data types but must have the same number of rows.

```
LastNames = {'Sanchez'; 'Johnson'; 'Li'; 'Diaz'; 'Brown'};  
Age = [38;43;38;40;49];  
Smoker = logical([1;0;1;0;1]);  
Height = [71;69;64;67;64];  
Weight = [176;163;131;133;119];  
BloodPressure = [124 93; 109 77; 125 83; 117 75; 122 80];
```

Create a table, T, as a container for the workspace variables. The table function uses the workspace variable names as the names of the table variables in T. A table variable can have multiple columns. For example, the BloodPressure variable in T is a 5-by-2 array.

```
T = table( LastName, Age, Smoker, Height, Weight, BloodPressure )
```

LastNames	Age	Smoker	Height	Weight	BloodPressure
{'Sanchez'}	38	true	71	176	124 93
{'Johnson'}	43	false	69	163	109 77
{'Li'}	38	true	64	131	125 83
{'Diaz'}	40	false	67	133	117 75
{'Brown'}	49	true	64	119	122 80

You can use dot indexing to access table variables. For example, calculate the mean height of the patients using the values in T.Height.

```
meanHeight = mean(T.Height)
```



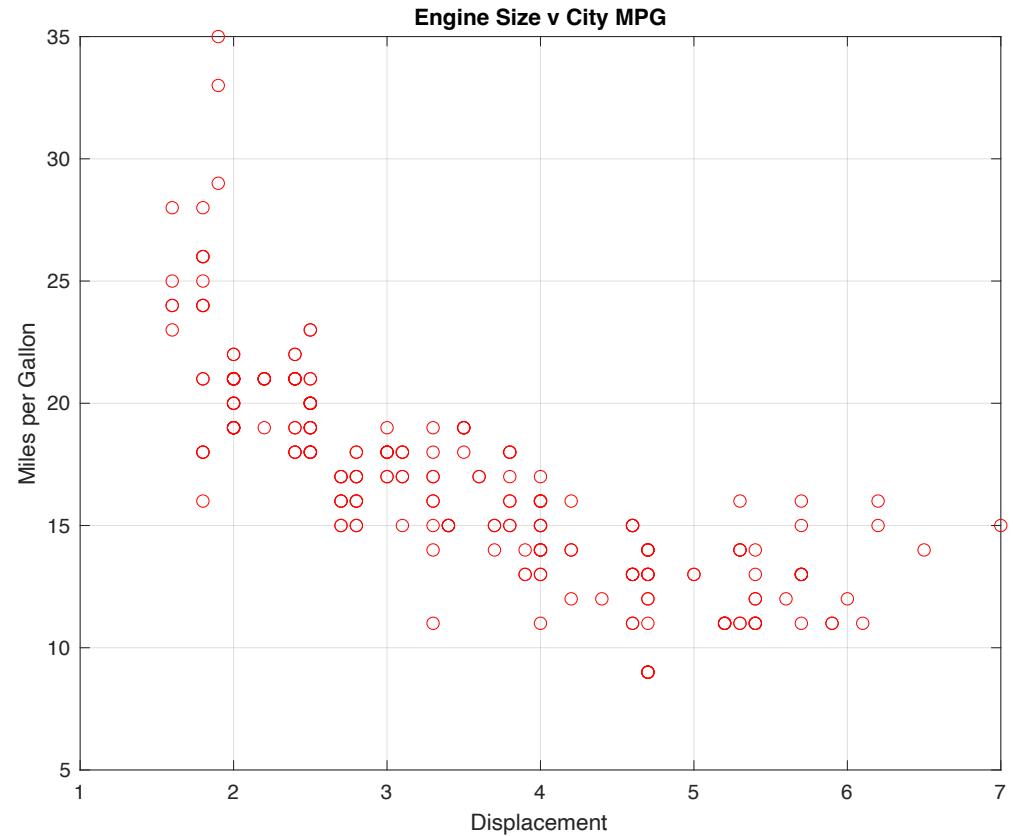
Exploring Data, first steps

- Generate a first graph to help answer the following question:
 - *Do cars with big engines use more fuel than cars with small engines*
- What might the relationship between engine size and fuel efficiency look like?

```
> mpg
# A tibble: 234 x 11
  manufacturer model      displ  year   cyl trans     drv   cty   hwy fl class
  <chr>        <chr>    <dbl> <dbl> <int> <chr>     <chr> <dbl> <dbl> <chr> <chr>
1 audi         a4          1.8  1999     4 auto(l5)   f       18     29 p   compact
2 audi         a4          1.8  1999     4 manual(m5) f       21     29 p   compact
3 audi         a4          2    2008     4 manual(m6) f       20     31 p   compact
4 audi         a4          2    2008     4 auto(av)   f       21     30 p   compact
```

Visualisation – using plot()

```
clear;  
  
mpg = readtable("mpg.xlsx");  
  
plot(mpg.displ,mpg.cty,"or");  
grid();  
title("Engine Size v City MPG");  
xlabel("Displacement");  
ylabel("Miles per Gallon");
```



Visualisation – combining variables

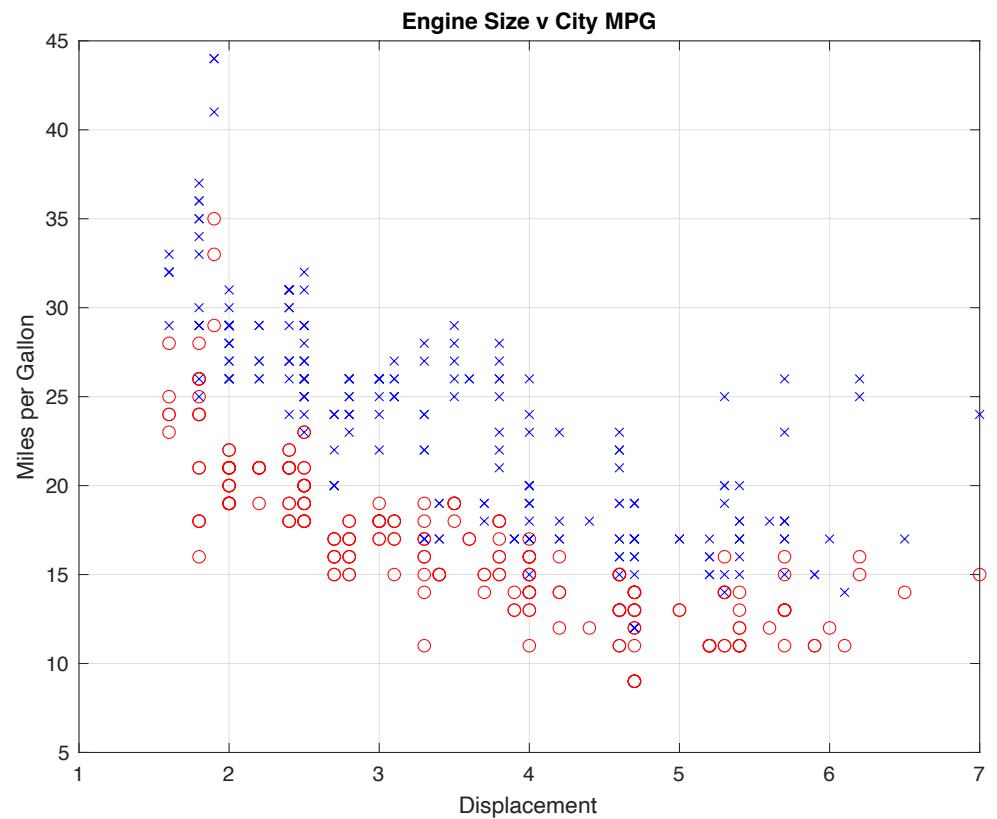
```
clear;

mpg = readtable("mpg.xlsx");

plot(mpg.displ,mpg.cty,"or");
hold on;
plot(mpg.displ,mpg.hwy,"xb");

grid();
title("Engine Size v City MPG");
xlabel("Displacement");
ylabel("Miles per Gallon");

hold off;
```



Variation and Histograms

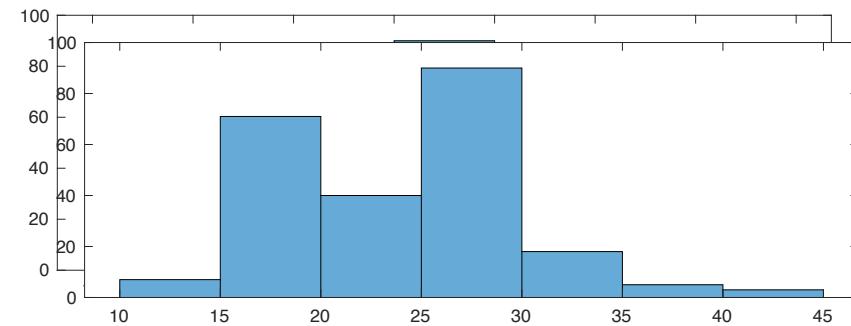
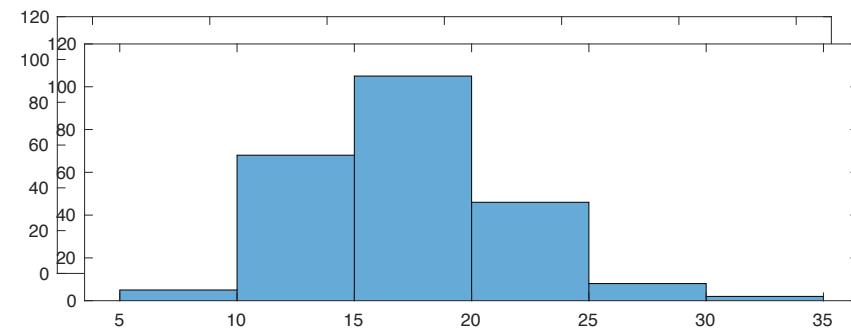
- **Variation** is the tendency of the values of a variable to change from measurement to measurement
- If you measure a continuous variable twice, you can get two different results
 - For example, temperature at 12.00 and temperature at 13.00
- Very variable has its own pattern of variation
- The best way to understand the pattern is to *visualize the distribution* of a variable's values.
- A variable is continuous if it can take on any of an infinite set of ordinal values
- Numbers are an example of a continuous variable
- To examine the distribution of a continuous variable, use a histogram

Variation - Histogram

- A histogram divides the x-axis into equally spaced bins, and uses the height of each bar to display the number of observations that fall into each bin
 - Tall bars show the common values of a variable, shorter bars show less common values
 - Places with no bars reveal values that were not seen in your data
-
- **Some useful questions**
 - Which values are most common? Why?
 - Which values are rare? Does it match your expectation?
 - Can you see any unusual patterns? What might explain them?

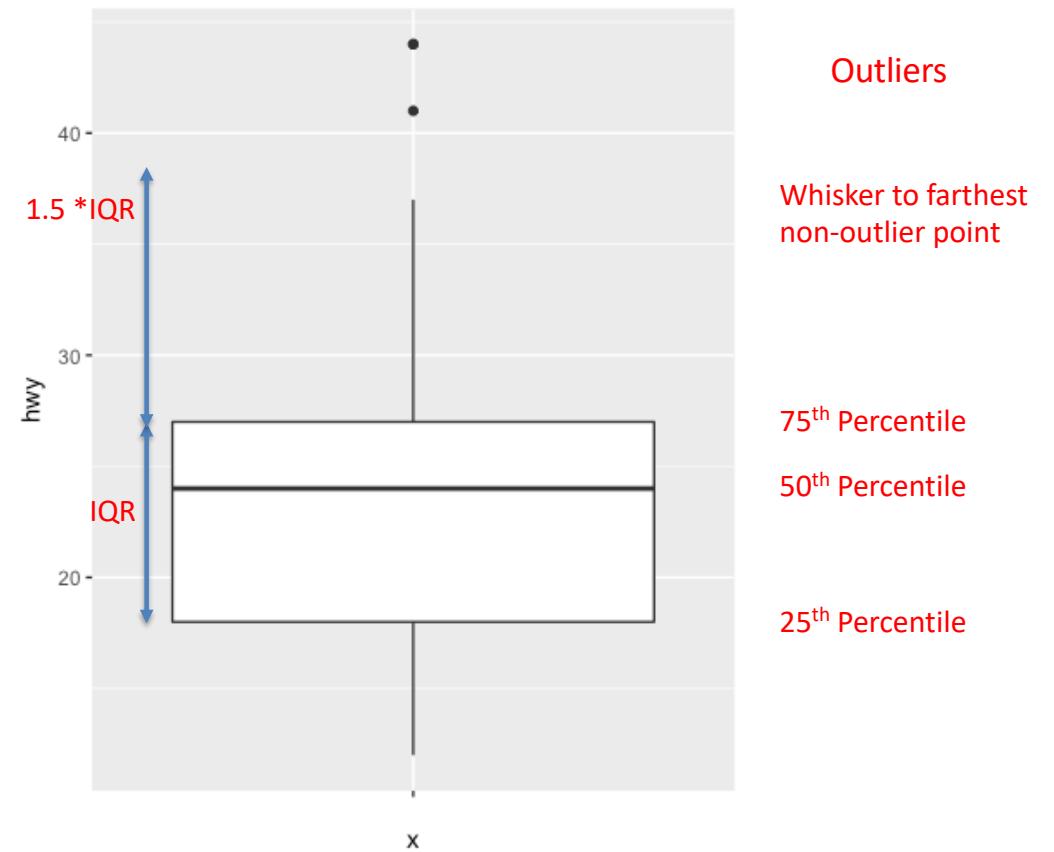
Example

```
clear;  
  
mpg = readtable("mpg.xlsx");  
  
subplot(2,1,1);  
histogram(mpg.cty, 'BinWidth', 5);  
subplot(2,1,2);  
histogram(mpg.hwy, 'BinWidth', 5);
```



Boxplot

- Display the distribution of a continuous variable broken down by a categorical variable
- Box that stretches from the 25th to 75th percentile a distance known as the interquartile range (IQR)
- Median in the middle of box
- Points outside more than 1.5 times the IQR from either edge of the box are displayed (outliers)
- Whisker extends to the farthest non-outlier point in the distribution

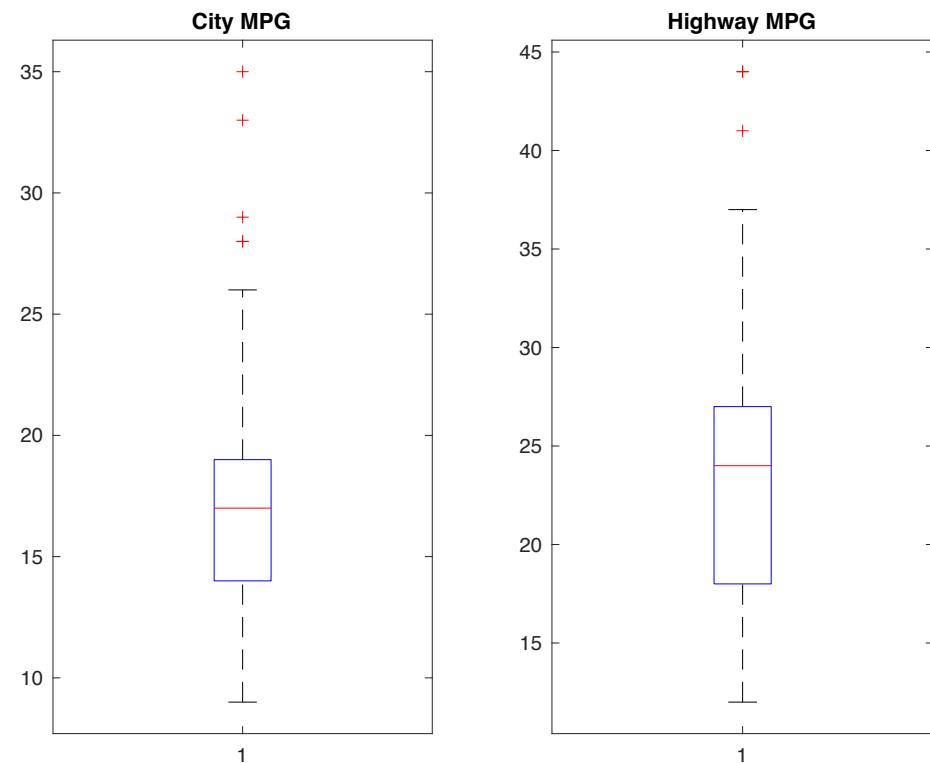


Displaying Boxplots

```
clear;

mpg = readtable("mpg.xlsx");

subplot(1,2,1);
boxplot(mpg.cty);
title("City MPG");
hold on;
subplot(1,2,2);
boxplot(mpg.hwy);
title("Highway MPG");
```



```
>> unique(mpg.class)
```

```
ans =
```

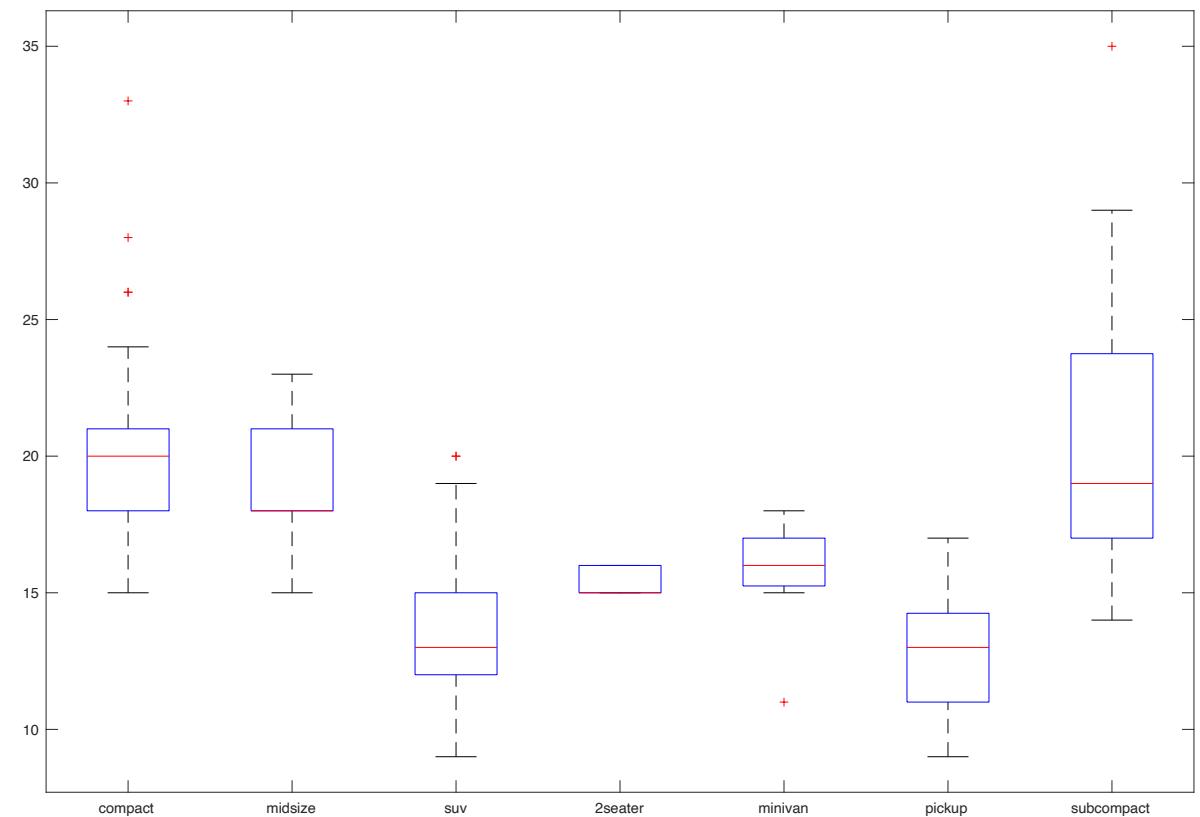
7×1 **cell** array

```
{'2seater' }  
{'compact' }  
{'midsize' }  
{'minivan' }  
{'pickup' }  
{'subcompact'}  
{'suv' }  
}
```

```
clear;
```

```
mpg = readtable("mpg.xlsx");  
  
boxplot(mpg.cty, mpg.class);
```

Using Categorical Variables



Filtering Data (using logical vectors)

- Tables can be filtered using logical vectors
- For example, filter all cars with displacement greater than 6.0

manufacturer	model	displ
{'chevrolet'}	{'corvette'}	6.2
{'chevrolet'}	{'corvette'}	6.2
{'chevrolet'}	{'corvette'}	7
{'chevrolet'}	{'k1500 tahoe 4wd'}	6.5
{'jeep'}	{'grand cherokee 4wd'}	6.1

```
clear;  
  
mpg = readtable("mpg.xlsx");  
  
lv = mpg.displ > 6.0;  
  
sub_mpg = mpg(lv,:);  
  
sub_mpg(:,1:3)
```

Challenge 10.1

- Write a script that will plot on the one scatter plot (in 2 different colours) the disp v cty for the following classes of car
 - compact
 - suv
- Use a filter-like operation within the loop for the 2 classes of car

```
>> unique(mpg.class)  
ans =  
7×1 cell array  
  
{'2seater' }  
{'compact' }  
{'midsize' }  
{'minivan' }  
{'pickup' }  
{'subcompact'}  
{'suv' }
```